Neural Networks for Structured Data

by

Kee Siong Ng

Department of Computer Science
The Australian National University

A thesis submitted to the
Faculty of Engineering and Information Technology
in partial fulfillment of the requirements for the degree of
Bachelor of Information Technology (Honours)

© Kee Siong Ng, November 2001

This work is affectionately dedicated to Pasiphae, my office computer
who spent many sleepless nights crunching numbers
to produce the results I report herein.

Except where otherwise indicated, this thesis is my own original work.

Kee Siong Ng
November, 2001

Abstract

Attribute-value language (AVL) is the current dominant knowledge representation formalism (KRF) in machine learning. While this form of KRF is sufficient for many learning tasks, it proves to be unnecessarily restrictive in several increasingly important application domains like bioinformatics and text-mining on structured documents. In these domains, the individuals have complex internal structures that cannot be easily captured using AVL. This provides motivation to extend current learning techniques to handle individuals of these more complicated data types.

This thesis presents an extension of artificial neural networks to handle structured data. We represent individuals using terms in a typed higher-order logic. This allows us to model a wide range of structured data, including tuples, sets, multisets, lists, trees, graphs, etc. A method to construct neural networks capable of learning functions mapping terms to real-valued vectors is presented. We also describe in detail an algorithm based on gradient descent suitable for training this new class of neural networks. Finally, experimental results on both artificial and real datasets are presented.

Acknowledgements

This thesis is the third biggest thing I have done in my life so far. The year-long journey towards its completion would have been boring and much tougher if not for the wonderful people around me, all of whom have at one stage or another, kindly lend me their helping hands. To the following individuals and institution, I say thank you.

For being my supervisors, I thank Distinguished Professor John Lloyd and Dr Lex Weaver. John, I have learned a lot from you in the past two years. Your wealth of knowledge has always been a source of inspiration. This work would not have been possible if not for your help and constant encouragements. Lex, thanks for the many good advice you gave me on a wide range of issues. I know you have a thesis to write as well, in addition to a million other responsibilities. I appreciate all the time you have given me.

For proof-reading draft manuscripts of this thesis, I thank John, Lex, Professor Rajeev Goré, Professor Eric McCreath, Dr Cheng Soon Ong, Dr Richard Walker, Dr Kee Wee Ng, Dr Agnes Boskovitz, Dr Edward Harrington and Dr Xiaobing Wu. Raj is undoubtedly the most prolific bug-tracker of them all; he found 10,534 bugs in a first draft of this thesis! John came in a close second; he managed to identify a total of 9,834 bugs in the three drafts that he read. Not far behind is Eric; he tracked down 8,523 bugs. The rest each identified at least 1,000 errors. I thank them all for their careful reading.

For helping me financially, I thank my parents, my brother, John, Dr Sigi Goode, Professor Weifa Liang and The Australian National University (ANU). My parents and brother supported me unconditionally through-out the last four years. John, Sigi and Weifa each gave me a job that I would have done for free. ANU gave me four years of free education. You are all perfect counter-examples to the no-free-lunch "theorem".

For constantly reminding me to have a life, I thank Ms Joanne Yin and Ms Linda Laohachai. Both of them "happily" listened to my many whinges when things were not going smoothly. I really appreciate that.

Finally, for many enjoyable lunch time discussions, I thank my fellow office mates Agnes, Cheng Soon, Edward, Dr Douglas Aberdeen, Dr Kerry Trentelman, Dr Nicolette Bonnette and Dr Paul Wong. Guys, I honestly believe P is a proper subset of NP, and I still think whale farming for large-scale milk production is an absurd idea!

Contents

Abstract i

Acknowledgements i

List of Figures iv

List of Tables vi

1 Overview 1
- 1.1 Introduction . . . 1
- 1.2 Motivations . . . 2
- 1.3 Basic Concepts . . . 3
  - 1.3.1 Machine Learning . . . 3
  - 1.3.2 Artificial Neural Networks . . . 4
  - 1.3.3 Approaches to Structured Data Processing . . . 6
- 1.4 The Problem . . . 8

2 Approximation 10
- 2.1 Representation of Individuals . . . 10
  - 2.1.1 Overview of Higher-order Logic . . . 10
  - 2.1.2 Basic Terms . . . 11
- 2.2 Term Reduction Networks . . . 13
- 2.3 Design Issues for Term Reduction Networks . . . 18
  - 2.3.1 Network Topology . . . 18
  - 2.3.2 The Labelling Function . . . 20

3 Computation 23
- 3.1 An Exercise in Function Optimisation . . . 23
  - 3.1.1 The Error Function . . . 24
  - 3.1.2 Optimisation Techniques . . . 25
- 3.2 Error Back-Propagation . . . 26
  - 3.2.1 Standard Error Back-Propagation . . . 26
  - 3.2.2 Back-Propagation Through Structure . . . 29

4 Results 33
4.1 Five Easy Pieces . . . 33
4.1.1 Tennis (Tuple) . . . 33
4.1.2 Keys (Set) . . . 35
4.1.3 The East-West Challenge (List) . . . 37
4.1.4 Interesting Trees (Binary Tree) . . . 41
4.1.5 Bongard 47 (Directed Graph) . . . 42
4.2 Three Less Easy Problems . . . 47
4.2.1 Binary Search Trees . . . 47
4.2.2 Musk . . . 48
4.2.3 The 2000-1 Predictive Toxicology Challenge . . . 52
4.3 An Analysis of the Results . . . 56

5 Related Work 58
5.1 Introduction . . . 58
5.2 Hinton's Reduced Descriptions . . . 58
5.3 Pollack's Recursive Auto-Associative Memory . . . 59
5.4 Labelling RAAM . . . 62
5.5 Folding Architecture . . . 63
5.6 Generalised Recursive Neurons . . . 65

6 Conclusion 67
6.1 Contribution of this thesis . . . 67
6.2 Future work . . . 68

A The (Simple) Mathematics of Back-propagation 69

B Sample Individuals in the BST Dataset 73

References 75

List of Figures

1.1 (a) A linear unit. (b) A non-linear unit. 4
1.2 (a) A feedforward neural network. (b) A recurrent neural network. 5
1.3 Molecule TR000 in PTC 2000-01 6

2.1 The relationships between terms, normal terms and basic terms. 14
2.2 Structure reduction network 15
2.3 Abstraction reduction network - alternative 1 15
2.4 Abstraction reduction network - alternative 2 16
2.5 Tuple reduction network 16
2.6 Neural network for the term (BNode (BNode Null 'A' Null) 'B' (BNode Null 'C' Null), $\{3, 10, 15\}$, [20.0, 14.0]) 19

3.1 Stochastic gradient descent version of standard error back-propagation. . . 27
3.2 The error surface of a simple network with one adjustable weight. 29
3.3 The stochastic gradient descent version of Back-propagation through Structure. 31

4.1 The tennis network component. Two nodes (excluding the bias node) were used in the hidden layer. The black nodes are bias nodes with input fixed at 1. All the nodes are numbered for referencing purposes. 35
4.2 The network components. Two nodes (excluding the bias node) were used in the hidden layer. The black nodes are bias nodes with input fixed at 1. (a) The *Key* component. (b) The $\{Key\}$ component. 37
4.3 Trains in Michalski's East-West Challenge. 38
4.4 The trains network components. One node (excluding the bias node) was used in the hidden layer. The black nodes are bias nodes with input fixed at 1. All the nodes are numbered for referencing purposes. (a) The $[Car]$ component. (b) The *Car* component. 40
4.5 The binary tree network component. Two nodes (excluding the bias node) were used in the hidden layer. The black nodes are bias nodes with input fixed at 1. All the nodes are numbered for referencing purposes. 42
4.6 Bongard: Examples in Class1 are on the left; those in Class2 are on the right. 43

4.7 Network components for the Bongard problem. Four nodes (excluding the bias node) were used in the hidden layer. The black nodes are bias nodes with input fixed at 1. All the nodes are numbered for referencing purposes. 45
4.8 The binary search tree network component. 48
4.9 The musk network components. 50
4.10 The {*Conformation*} network component. 52
4.11 Sample molecules in The 2000-1 Predictive Toxicology Challenge. 54

5.1 (a) A simple binary tree. (b) A typical binary tree data structure on von Neumann machines implemented using pointers. (c) A possible connectionist representation. 59
5.2 Single 2k-k-2k network composed of both the encoding and decoding network. 60
5.3 To learn both the encoding and decoding mechanisms for the binary tree on top of the figure, the 2k-k-2k network needs to learn the four auto-associations shown. Each rectangle is a K-tuple. 61
5.4 The general network architecture of a LRAAM. 62
5.5 The LRAAM-Feedforward network architecture capable of performing classification tasks on terms. Note the double arrow between the input and hidden layers. 63
5.6 The generic form of a folding architecture. 63
5.7 The encoder unfolded by the structure $f(X, g(a, Y))$. 64
5.8 Neuron models from different input domains. The standard neuron is suitable for the processing of unstructured patterns, the recurrent neuron for the processing of sequences of patterns, and the generalised recursive neurons for the processing of structured patterns. 65
5.9 A graph and its corresponding encoding network. There is a generalised recursive neuron for every node in the graph. The topology of the network looks similar to the topology of the graph. 66

List of Tables

4.1 The weight solution for tennis. 35
4.2 The weight solution for the Keys problem. 37
4.3 The weight solution for the East-West problem. 41
4.4 The weight solution for the binary tree component. 42
4.5 The weight solution for the Bongard experiment. 46
4.6 Summary of musk-1 results : Accuracy vs Hidden nodes. 50
4.7 Summary of musk-2 results : Accuracy vs Hidden nodes. (early stopping without validation set) 50
4.8 Known results for the musk problem. 51

Chapter 1

Overview

There is nothing more difficult to take in hand,
more perilous to conduct, or more uncertain in its success,
than to take the lead in the introduction of a new order to things.
N. Machiavelli

1.1 Introduction

The three topics of interest in artificial neural networks are approximation, computation, and estimation. *Approximation* is concerned with the representational properties of neural networks. A network with a certain topology has associated with it a class of functions implementable by that network. This can be thought of as its capacity. Given a set of training examples and some background information on the target function, on the approximation side, we ask what is the best (or, less ambitiously, a good) network architecture to use to try to model the target function? A good choice is one that implements a class of functions that contains within it the target function, yet which is sufficiently constrained that the risk of overfitting is minimal. *Computation* is related to the algorithmics side of neural network learning. Having chosen a good architecture, we ask how can we best make use of the training data available to compute the desired function? Problems of interest in this area include finding efficient and elegant algorithms to solve optimisation problems, and working out ways to navigate "treacherous" error function surfaces. Finally, after having worked through the first two types of problems and arriving at a solution, we are interested in a statistical *estimation* of how close our solution really is to the (unknown) true target function, *i.e.,* how well is it likely to perform on unseen data? Results on this class of problems are usually in the form

$$\text{true error} \leq \text{empirical error} + \text{complexity penalty},$$

where empirical error is the measured error on the training data, and complexity penalty is a function of the capacity of the network and the statistical confidence of the bound.

The current work is primarily concerned with the approximation issues of neural network learning. We build on earlier attempts on adaptive structural processing exemplified by the works of Pollack [Pol90], Sperduti *et al.* [SSG95, SS97] and Goller *et al.* [GK96, Gol97, Ham96], and introduce a way to extend feedforward neural networks to learn mappings from terms in a higher-order logic to real-valued vectors. The use of higher-order terms allows us to model a wide range of structured data that includes tuples, sets, multisets, lists, trees, graphs, etc. This extra richness in knowledge representation allows us to tackle problems hitherto difficult to solve using neural networks. We called this class of networks derived from logical terms *term reduction networks.* Some parallel work on the computational issues of training term reduction networks are also explored.

Thesis Organisation The remainder of this chapter presents the motivations, terminology, and other background information relevant to this thesis. Chapter two describes term reduction networks in detail. This is followed by a discussion of an algorithm based on gradient descent for training term reduction networks in chapter three. Chapter four presents some experimental results. Chapter five summarises some related work in the literature. Finally, Chapter six concludes.

1.2 Motivations

Attribute-value language (AVL) is the current dominant knowledge representation formalism (KRF) in machine learning. Using this scheme, individuals in application domains are identified and differentiated based on a selection of their attributes (or features), and represented simply as tuples of constants. We call tuples so constructed the *feature vectors* of the individuals. While this form of KRF is sufficient for many learning tasks, it proves to be unnecessarily restrictive in several increasingly important application domains where the individuals have complex internal structures that cannot be adequately captured using AVL. Examples include bioinformatics (*e.g.,* gene and protein analysis), chemoinformatics (*e.g.,* quantitative structure-property relationship (QSPR) and quantitative structure-activity relationship (QSAR) type problems), information retrieval (*e.g.,* text mining on XML and HTML documents), automated reasoning (*e.g.,* handling of logical terms), etc. In the words of Quinlan [Qui96],

> "Data may concern objects or observations with arbitrarily complex structure that cannot be captured by the values of a predetermined set of attributes."

This provides strong motivation to extend existing machine learning techniques to handle individuals of these more complicated data types.

The need for a richer representation language is well recognised by many other authors, see for examples [Qui96, PK92, BGCL01], but a general consensus on what is the best formalism has not been reached. The majority view seems to be in favour of a first-order language. This line of thought has its root in the popularity of logic programming in the AI community. The successes of Quinlan *et al.* [Qui90, PK92, Qui96], Muggleton and De

Raedt *et al.* [MB88, MR94, Mug95, BR98] and many others who follow in the application of this formalism make it all the more persuasive.

In the last few years, a credible alternative in the form of higher-order languages has emerged, see for examples [FGCL98, Fla00, BGCL01]. Endorsement of this formalism comes mainly from people working in functional and functional logic programming. Their primary argument is that the strong typing system and high-order facilities of higher-order logic, unavailable in first-order logic, are highly desirable features of a good KRF, and that the additional complexity of a higher-order language is a fair price to pay for these linguistic and mathematical niceties. This is the viewpoint we take in this thesis.

1.3 Basic Concepts

In this section, I introduce the basic concepts that are relevant to the subject matter of the present work.

1.3.1 Machine Learning

Machine learning is the study of concept learning algorithms that improve automatically through experience. It is an inherently interdisciplinary field, built on concepts from artificial intelligence, probability and statistics, information theory, logic and philosophy, control theory, psychology, neurobiology, and many other fields [Mit97]. Typical applications of machine learning include hand-written character recognition, speech processing, email filtering, automated reasoning and planning, robot navigation, chess playing, etc.

Within machine learning, there are two separate subfields - supervised learning and unsupervised learning. The current work goes under the general framework of supervised learning. A typical learning task is posed in the following form:

> Given a set $\{\langle d_i, t_i\rangle \mid 1 \leq i \leq N\}$ of training examples, where each d_i represents an individual in the application domain, and t_i its label provided by a teacher, find a hypothesis that can 1) explain the relationship between the individuals and their labels; and 2) be used to predict the labels of previously unseen individuals with good accuracy.

In the traditional AVL setting, the d_i's are constrained to be feature vectors. In the current work, we consider problems of a more general nature, where the d_i's are higher-order terms that directly captures the (potentially complex) internal structures of the individuals. The type of the labels is usually either real numbers or a finite set of integers. In the former case, the target function is likely to be a regression function. In the latter case, it is likely to be a classification function. In some cases, additional background information about the learning task may be provided. Additional constraints may also be placed on the exact form the hypothesis can take.

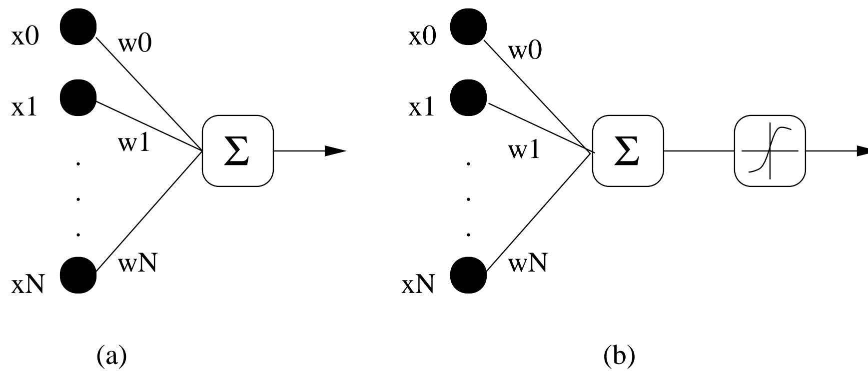


Figure 1.1: (a) A linear unit. (b) A non-linear unit.

1.3.2 Artificial Neural Networks

The artificial neural network (ANN) model is an important mathematical learning model that is widely used in fields as diverse as psychology, mathematics, neural science and computer science [Hay99]. In machine learning, they are used primarily as a tool to perform function approximation from examples.

An artificial neural network is a network of simple processing units with weighted connections between them. In many respects, ANNs are like their biological counterparts, if only significantly simpler. ANNs can learn through adaptation, by systematically adjusting their weights over time, and they exhibit complex and interesting behaviours through dynamic and parallel interactions between their processing units.

Processing units The simple processing units, or neurons as they are commonly called, can come in two different forms: linear units and non-linear units. These are shown in Figure 1.1. Linear units are used to produce networks that approximate linear functions. They produce as outputs a linear transformation of their inputs using the following function

$$o = \sum_i w_i \cdot x_i$$

which simply computes a weighted sum of the input values. Non-linear units, on the other hand, are used to produce networks that can approximate non-linear functions. They differ from linear units in that the weighted sum of the input values are further transformed using a nonlinear *squashing function*, *i.e.,* they implement functions of the form

$$o = \sigma(\sum_i w_i \cdot x_i)$$

where σ is a squashing function that maps large input values to small output values within a certain range, say between -1 and 1. A common choice of σ is the logistic or sigmoid

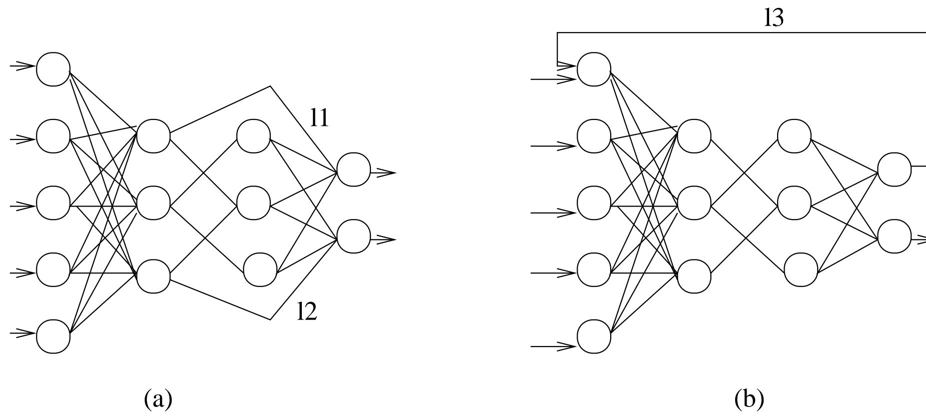


Figure 1.2: (a) A feedforward neural network. (b) A recurrent neural network.

function (shown below left)

$$\sigma(y) = \frac{1}{1+e^{-y}} \qquad \frac{d\sigma(y)}{dy} = \sigma(y) \cdot (1 - \sigma(y))$$

which has the very nice property that its derivative can be expressed in terms of its output (shown above right). One popular alternative to the logistic function is the hyperbolic tangent function tanh. In both the units shown in Figure 1.1, the input x_0 is a *bias input* added on top of the original input vector. Its value is always fixed at 1.

Network topologies The most common types of network topologies are feedforward networks and recurrent networks. Examples of the two are shown in Figure 1.2. Both networks shown in the diagram are three layered networks with two hidden layers and an output layer. By convention, the input layer is not counted because neurons in this layer perform no computation. It is customary to number the layers from the input layers to the output layer. In this case, the input layer is layer 0, the first hidden layer is layer 1, the second hidden layer is layer 2, and the output layer is layer 3. Nodes in the networks above can be linear units or non-linear units or a combination of both.

In feedforward networks, all the links are limited to be forward links, *i.e.,* a neuron in layer L is only allowed to connect to neurons in layers higher than L. Cross-layer forward links are allowed; $l1$ and $l2$ are examples of such links. Neurons may or may not connect to all the neurons in the next layer, as shown in the connections between layer 0 and layer 1, which is completely connected, and between layer 1 and 2, which is not. A feedforward network where every neuron in layer L is connected to every neuron in layer $L+1$ is called a *fully connected network*. This is the most common topology in use.

In recurrent networks, cyclical connections are admissible, *i.e.,* neurons can connect to other neurons in the preceding layers. $l3$ is an example of a backward connection. Such cyclical links can be thought of as feedback loops, where the output of a neuron is used

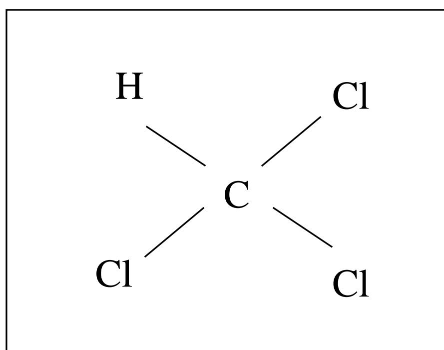


Figure 1.3: Molecule TR000 in PTC 2000-01

to feed back to one or more of its inputs. This makes the output of the neuron not just a function of the inputs, but a function of the inputs and its own earlier outputs. Recurrent networks are in general much harder to analyse compared to feedforward networks, but they have many interesting properties which make them delightful subjects of study.

1.3.3 Approaches to Structured Data Processing

For those working in a strictly AVL setting, learning on structured data can only be done after a data preprocessing stage, in which the complex individuals are transformed into feature vectors through a feature construction process. A paragraph from [HK01, p.396-397] explains this process in the context of data mining succinctly:

> "A complex structure-valued attribute may contain sets, tuples, lists, trees, records, and so on, and their combinations, where one structure may be *nested* in another at any level. In general, a structure-valued attribute can be generalised in several ways, such as (1) generalizing each attribute in the structure while maintaining the shape of the structure, (2) flattening the structure and generalizing the flattened structure, (3) summarizing the low-level structures by high-level concepts or aggregation, and (4) returning the type or an overview of the structure."

To illustrate this, consider molecules in a bioinformatics application. Figure 1.3 shows the molecule TR000 in the 2000-1 Predictive Toxicology Challenge [PTC01]. How can we construct a feature vector given such a molecule? We can, for instance, extract useful figures like the number of atoms of a particular type (like carbon, chlorine, hydrogen, etc) it has; the total number of atoms it has; the number of bonds of a certain type (like single bonds, double bonds, triple bonds, etc) it has; its molecular weight; etc. In addition, we can also introduce higher level concepts like hasAromaticRings, hasHalogens, hasBenzeneRings and so on to characterise the presence or otherwise of certain molecular substructures.

As we can see from the simple example above, feature construction is a tedious and manual task that requires good domain knowledge for success. It is also an inherently

problematic task, as depending on the experience and knowledge of the human experts doing the job, the resulting feature vectors may or may not contain the essential data for learning to proceed. This process is essentially a black art.

As mentioned previously in Section 1.2, there are many approaches to overcome the limitations of AVL. Dialects of first-order logic are common choices. The use of a first-order language allows us to model and reason about complex structured individuals directly. For example, the molecule shown in Figure 1.3 can be represented in Prolog [Bra90], which is first order, as follows:

```
atom('TR000','TR000_1'). element('TR000_1',cl).
...
atom('TR000','TR000_5'). element('TR000_5',h).
bond('TR000','TR000_1_2').
connected('TR000_1','TR000_2','TR000_1_2').
connected('TR000_2','TR000_1','TR000_1_2').
bond_type('TR000_1_2',-).
...
bond('TR000','TR000_2_5').
connected('TR000_2','TR000_5','TR000_2_5').
connected('TR000_5','TR000_2','TR000_2_5').
bond_type('TR000_2_5',-).
```

This is the commonly used "flattened" table-of-facts representation. An alternative way, the so-called individuals-as-terms approach, can be used as well. Under this scheme, each individual is represented as one single term. For example, the same molecule above can be represented as follows,

```
TR000 = graph([cl1, c2, cl3, cl4, h5],
              [e(cl1,c2,-),e(c2,cl3,-),e(c2,cl4,-),e(c2,h5,-)])
```

where the first list denotes the atoms of the molecule, and the second list denotes the connections between the atoms. In both representations, all the structural properties of the molecule are captured compactly without any loss of information.

Higher-order logic can also be used to represent structured data. A way to represent molecule TR000 in higher-order logic is as follows:

```
TR000 = ({(1,cl), (2,c), (3,cl), (4,cl), (5,h)},
         {(1,2,-), (2,3,-), (2,4,-), (2,5,-)}).
```

From that, one can see this representation has some advantages over its first-order counterpart. It is closer to the mathematical definition of a graph, which is a tuple with two sets, the first of which is the set of the vertices, and the second the set of edges in the graph. The notion of sets cannot be captured quite as naturally in first-order logic without the use of some abstract data types. In the first-order individuals-as-terms approach above, a list is used in place of a set. Strictly speaking, they are rather different. In lists, the

order is important, whereas in sets, the order is not. It can be envisaged that such subtle differences in the exact type of the individuals can have significant ramifications on the learnability and final form of the hypothesis concept.

1.4 The Problem

We conclude the chapter with a problem statement.

To represent structured data, we use a special class of terms in a typed higher-order logic [Llo01] called *basic terms*. The exact definition of basic terms is given in Section 2.1. For the purpose of this discussion, they can be thought of as terms admissible in a functional programming language like Haskell [Has], with additional syntactic provision for higher-order constructs like sets and multisets. Denoting the set of all basic terms by $\mathfrak{B}$, the problem this thesis attempts to solve is as follows:

> Given a set of individuals in the application domain all of type α, and denoting the set of all basic terms of type α by $\mathfrak{B}_\alpha$,
>
> 1. find a class of feedforward neural networks that can approximate functions with the following signature $f : \mathfrak{B}_\alpha \to \mathbb{R}^m$.
> 2. devise an algorithm that can train the class of feedforward neural networks in 1 and produce solutions with good predictive power.

Chapter Notes

The grouping of major research efforts on supervised learning into the three themes *approximation*, *computation* and *estimation*, as described in the introductory text of this chapter was first made in [AB99]. In the current work, we do not touch on estimation issues in any depth. This does not do justice to the importance of the subject matter. In actual fact, estimation is the current dominant research theme in machine learning, and almost all recent significant advances in machine learning are founded on and motivated by improved theoretical understanding of the estimation aspects of learning. Readers are referred to [CST00] and [AB99] for an introduction to this very important subject.

Readers interested in knowing more about machine learning are referred to [Mit97], a most authoritative introductory textbook on the subject. For an introductory text on inductive logic programming (ILP), a subfield of machine learning most closely related to the subject matter of the current work, see [Mug99]. For an account of the logical foundations of ILP, see [NCdW97]. Literally thousands of books have been written on various aspects of artificial neural networks. From a machine learner's perspective, perhaps the most well-written of those (and certainly the most cited) is [Bis95]. Other good references include [RM99], an accessible (meaning not very mathematical!) and practical introductory book on feedforward neural networks; and [Hay99], a comprehensive textbook that covers most aspects of neural network learning, including chapters on support vector machines and neurodynamic programming (reinforcement learning).

A year spent in artificial intelligence is enough to make one believe in God.
Alan J. Perlis (Epigrams of Programming)

If the human mind was simple enough to understand,
we'd be too simple to understand it.
Emerson Pugh

Chapter 2

Approximation

Although this may seem a paradox,
all exact science is dominated by the idea of approximation.
Bertrand Russell

In this chapter, we describe in detail a class of feedforward neural networks capable of approximating functions from $\mathfrak{B}_\alpha$ to $\mathbb{R}^m$, for some type α and a positive integer m. Section 2.1 describes the symbolic representation of individuals in the form of higher-order terms. Section 2.2 presents a systematic approach for constructing feedforward networks from higher-order terms. We conclude with a discussion of some important issues in the design of term reduction networks for real applications.

2.1 Representation of Individuals

The setting we use for representing individuals is a polymorphically typed higher-order logic. This provides a variety of data types for representing individuals. The full glory of the logic and its applications to the design of declarative languages and machine learning can be found in [Llo01], [Llo95], [Llo99], [BGCL00] and [NLS01]. The following is a summary of the relevant parts of the logic. With permission from the author, some formal definitions from Section 3 of [Llo01] are reproduced here.

2.1.1 Overview of Higher-order Logic

In brief, the logic is based on Church's simple theory of types [Chu40] with several extensions, the most significant of which is the polymorphic extension to the type system. The alphabet of the logic consists of four sets, a set $\mathfrak{T}$ of type constructors, a set $\mathfrak{P}$ of parameters (type variables), a set $\mathfrak{C}$ of constants, and a set $\mathfrak{V}$ of variables. Always included in the set $\mathfrak{T}$ are type constructors $\mathbf{1}$ and Ω both of arity 0. $\mathbf{1}$ is the type of some distinguished singleton set and Ω is the type of the booleans. The types of the logic are built up from the set of type constructors and the set of parameters using the symbols $\rightarrow$ and $\times$. The

former is used to construct function types, the latter, product types. A type is closed if it contains no type variables. The set of all closed types is denoted by $\mathfrak{S}^c$.

$\mathfrak{C}$ is the set of constants of various types. The two constants $\top$ (true) and $\bot$ (false) are always included in it. One can distinguish between two kinds of constants, *data constructors* and *functions*. In a knowledge representation context, data constructors are used to represent individuals. In a programming language context, data constructors are used to construct data values whereas functions are used to compute on data values. Functions have definitions, data constructors do not. In the semantics of the logic, which is a Henkin model [Hen50], the data constructors are used to construct models [Llo01].

The terms of the logic are the terms of the typed λ-calculus, which are formed in the usual way by abstraction and application (or juxtaposition) from constants and variables. Products of terms can also be formed using a tuple-forming notation $(\dots)$. The set of all terms is denoted by $\mathfrak{L}$ (for language).

2.1.2 Basic Terms

A suitable class of closed terms called basic terms are used to represent individuals in our application domains. Before giving the definition of basic terms, we need a few more concepts. These will be developed in stages as needed. First, we identify a subset of terms called normal terms that are suitable for knowledge representation. We point out the undesirable non-uniqueness of normal terms and describe a way to solve this problem. Finally, we define basic terms as the distinguished representatives of equivalence classes of normal terms.

For reasons which will become clear, we need the concept of a default term for each type. For example, we can use 0 as the default term for integers, false for the booleans, etc. The set of default terms is denoted by $\mathfrak{D}$.

Normal Terms We now give the definition of normal terms.

Definition. The set of *normal terms*, $\mathfrak{N}$, is defined inductively as follows.

1. If C is a data constructor having signature $\sigma_1 \to \cdots \to \sigma_n \to (T\ a_1 \dots a_k)$ and $t_1, \dots, t_n \in \mathfrak{N}$ $(n \geq 0)$ such that $C\ t_1 \dots t_n \in \mathfrak{L}$, then $C\ t_1 \dots t_n \in \mathfrak{N}$.

2. If $t_1, \dots, t_n \in \mathfrak{N}$, $s_1, \dots, s_n \in \mathfrak{N}$ $(n \geq 0)$, $s_0 \in \mathfrak{D}$ and
$$\lambda x.\mathit{if}\ x = t_1\ \mathit{then}\ s_1\ \mathit{else}\ \dots\ \mathit{if}\ x = t_n\ \mathit{then}\ s_n\ \mathit{else}\ s_0 \in \mathfrak{L},$$
then
$$\lambda x.\mathit{if}\ x = t_1\ \mathit{then}\ s_1\ \mathit{else}\ \dots\ \mathit{if}\ x = t_n\ \mathit{then}\ s_n\ \mathit{else}\ s_0 \in \mathfrak{N}.$$

3. If $t_1, \dots, t_n \in \mathfrak{N}$ $(n \geq 0)$ and $(t_1, \dots, t_n) \in \mathfrak{L}$, then $(t_1, \dots, t_n) \in \mathfrak{N}$.

Normal terms formed from Part 1 of the definition through application are called *normal structures* and always have a type of the form $T\alpha_1 \ldots \alpha_n$. Thus, we obtain atomic individuals like natural numbers, integers, etc, from data constructors with 0 arity, and more complicated structures like lists, trees from data constructors with non-zero arities.

Normal terms formed from Part 2 of the definition through lambda abstraction are called *normal abstractions*. They always have a type of the form $\beta \rightarrow \gamma$. This class of normal terms are essentially a form of finite lookup tables, where the value for element t_n is s_n. Now we see why default terms are needed. In a finite lookup table, only values for a (small) set of elements are usually defined. To implement a complete lookup table defined for all elements of a certain type, we can simply define all elements which are not explicitly specified as having the default value. The lambda function definition is mathematically very neat, allowing one to model sets, multisets and similar data types intensionally.

To see how this can be done, consider the set $\{1, 2\}$. In higher-order logic, sets can be viewed as predicates. For the given set, we can represent it intensionally using the term

$$\lambda x.\text{if } x = 1 \text{ then } \top \text{ else if } x = 2 \text{ then } \top \text{ else } \bot. \tag{2.1}$$

Similarly, a multiset with 4 occurrences of A and 23 occurrences of B can be represented intensionally as the term

$$\lambda x.\text{if } x = A \text{ then } 4 \text{ else if } x = B \text{ then } 23 \text{ else } 0.$$

Part 3 of the definition simply states that one can form a tuple from normal terms to obtain another normal term. Terms formed in this way are called *normal tuples* and they always have a type of the form $\alpha_1 \times \cdots \times \alpha_n$.

One problem with the use of normal terms as the knowledge representation language is that normal abstractions are not unique, *i.e.,* there are syntactically different normal terms which mean the same thing. Going back to the example $\{1, 2\}$ above, we can see that the following two function definitions, though syntactically different, are semantically equivalent to the one given earlier.

$$\lambda x.\text{if } x = 2 \text{ then } \top \text{ else if } x = 1 \text{ then } \top \text{ else } \bot \tag{2.2}$$

$$\lambda x.\text{if } x = 2 \text{ then } \top \text{ else if } x = 1 \text{ then } \top \text{ else if } x = 3 \text{ then } \bot \text{ else } \bot \tag{2.3}$$

To understand the way to fix this undesirable property of normal abstractions, we need to go no further than the two illustrative function definitions above. Function definition 2.2 is different from function 2.1 only in the order of the subterms of the form $x = y$. Function definition 2.3, in addition to the difference in order, contains redundant declarations. The intuitive idea is to devise a *regularise* procedure which, when given syntactically different but semantically equivalent normal terms of a particular type, is able to *reduce*, through syntactic manipulation, the different terms into identical ones. For example, given function 2.2, it will try to re-order the subterms. Given function 2.3, it will first remove redundant declarations and then re-order the subterms.

To devise such a procedure, we need to have more mathematical tools. Firstly, we need to define a strict total order $<$ on the normal terms, which can then be used to order

the elements of normal abstractions. Secondly, we need to define an equivalence relation $\equiv$ on normal terms of a certain type, which can then be used to identify and remove redundant elements of a function definition. Obviously, the equivalence relation needs to be defined in a way such that normal terms of a certain type denoting the same individual belong to the same equivalence class. We can then pick out one representative term from each equivalence class and use these as the knowledge representative language. Given a normal term, it is the unique representative term of the equivalence class in which the term belongs that our *regularise* procedure will return. The set of all such unique representative terms is called the basic terms, the formal definition of which is given in the following paragraph. The reader is referred to Section 3 of [Llo01] for a detailed formalisation of the ideas presented in this section.

Basic Terms

Definition. The set of *basic terms*, $\mathfrak{B}$, is defined inductively as follows.

1. If C is a data constructor having signature $\sigma_1 \to \cdots \to \sigma_n \to (T\ a_1 \ldots a_k)$ and $t_1, \ldots, t_n \in \mathfrak{B}$ $(n \geq 0)$ such that $C\ t_1 \ldots t_n \in \mathfrak{L}$, then $C\ t_1 \ldots t_n \in \mathfrak{B}$.

2. If $t_1, \ldots, t_n \in \mathfrak{B}$, $s_1, \ldots, s_n \in \mathfrak{B}$, $t_1 < \ldots < t_n$, $s_i \notin \mathfrak{D}$, for $1 \leq i \leq n$ $(n \geq 0)$, $s_0 \in \mathfrak{D}$ and
$$\lambda x.\mathit{if}\ x = t_1\ \mathit{then}\ s_1\ \mathit{else}\ \ldots\ \mathit{if}\ x = t_n\ \mathit{then}\ s_n\ \mathit{else}\ s_0 \in \mathfrak{L},$$
then
$$\lambda x.\mathit{if}\ x = t_1\ \mathit{then}\ s_1\ \mathit{else}\ \ldots\ \mathit{if}\ x = t_n\ \mathit{then}\ s_n\ \mathit{else}\ s_0 \in \mathfrak{B}.$$

3. If $t_1, \ldots, t_n \in \mathfrak{B}$ $(n \geq 0)$ and $(t_1, \ldots, t_n) \in \mathfrak{L}$, then $(t_1, \ldots, t_n) \in \mathfrak{B}$.

The basic terms from Part 1 of the definition are called *basic structures*, those from Part 2 are called *basic abstractions*, and those from Part 3 are called *basic tuples*.

Figure 2.1 shows diagrammatically the relationships between terms, normal terms and basic terms. Normal terms is a proper subset of terms. Within the set of normal terms, there are further subsets whose elements belong to the same equivalence class. Within each equivalence class, one unique representative term can be chosen. These are shown as asterisks in the diagram. The set of all asterisks is the set of basic terms.

2.2 Term Reduction Networks

I now describe a way to construct feedforward neural networks that can learn a function $f : \mathfrak{B}_\alpha \to \mathbb{R}^m$, for some type α and a positive integer m, from a set of training examples. We call networks so constructed *term reduction networks*.

The intuitive idea behind term reduction network is rather simple. Given a basic term of a certain type representing an individual in the application domain, we build up a neural network for it in a bottom up fashion in much the same way as we build up a syntax tree for

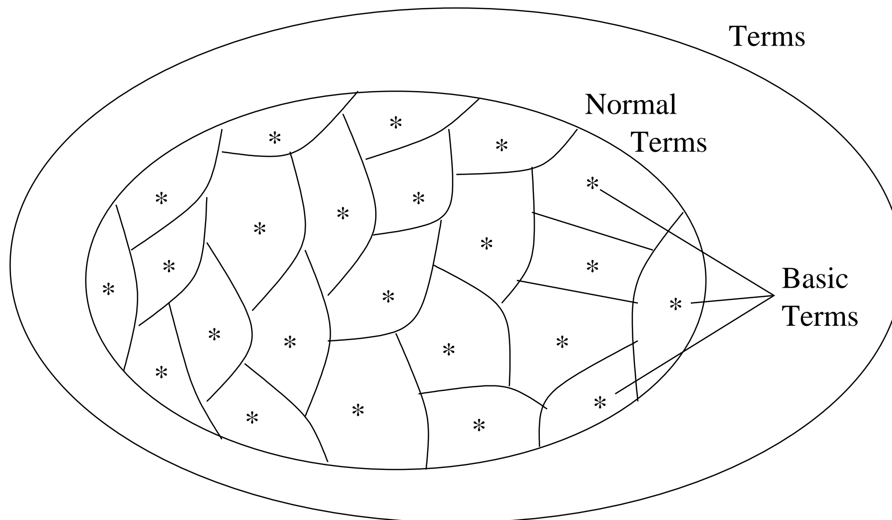


Figure 2.1: The relationships between terms, normal terms and basic terms.

it. The resulting network is composed of what we call reduction networks. They correspond to the subterms. Reduction networks are described next. This is followed by a description of the process of constructing neural networks from them.

The Basic Building Blocks

As can be seen from Section 2.1, a basic term is either a basic structure, a basic abstraction or a basic tuple. There is a class of reduction network for each of the three. These will be examined in turn. Before that, we need to introduce a static labelling function $L : \mathfrak{C} \rightarrow \mathbb{R}^n$ which maps constants in the logic to real-valued vectors.

Basic structures are not unlike the labelled trees of [GK96] and [Ham96]. The reduction network for basic structures looks just like the encoding part of folding architecture (see [GK96, KG96]). Figure 2.2 shows the structure reduction network for a data constructor C of arity n. Given the reductions of the data constructor C and its n arguments, the network produces as its output an element of $\mathbb{R}^p$. The reduction of C is performed by L. The reductions of the arguments are the outputs of other reduction networks. In applications, there is a structure reduction network for each data constructor having arity > 0. For data constructors with arity 0 (for example, integers, real numbers, etc), a direct reduction is used instead. This is again provided by the labelling function L.

The task of designing a reduction network for basic abstraction is not a straightforward one. This is not surprising, as the additional representation power provided by basic abstraction is what sets our knowledge representation formalism apart from the rest. From Section 2.1, we see that basic abstractions can be most straightforwardly understood as finite look-up tables. Typical instances of it in the context of knowledge representation are

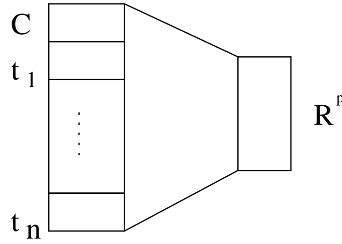


Figure 2.2: Structure reduction network

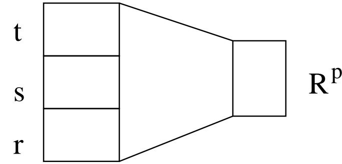


Figure 2.3: Abstraction reduction network - alternative 1

sets and multisets. There are many different ways to implement reduction networks for basic abstractions. One possibility is shown in Figure 2.3. To see how that is done, recall that basic abstractions have the form

$$\lambda x.\text{if } x = t_1 \text{ then } s_1 \text{ else } \ldots \text{ if } x = t_n \text{ then } s_n \text{ else } s_0 \in \mathfrak{B}_{\alpha \to \beta}$$

where $t_1, \ldots, t_n \in \mathfrak{B}_\alpha, s_1, \ldots, s_n \in \mathfrak{B}_\beta, n \geq 0, s_0 \in \mathfrak{D}$. We can reformulate the abstraction in the following equivalent form

$$\lambda x.\text{if } x = t_1 \text{ then } s_1 \text{ else } (r\ x)$$

where r is

$$\lambda x.\text{if } x = t_2 \text{ then } s_2 \text{ else } \ldots \text{ if } x = t_n \text{ then } s_n \text{ else } s_0.$$

By repeatedly rewriting basic abstractions into the new form and conjoining the resulting abstraction reduction networks together, we end up with an acceptable reduction network for basic abstraction. Note that for the special case where r becomes the identity function $\lambda x.s_0$, we use the labelling function L to obtain a direct reduction of s_0 in place of another abstraction reduction network.

One major drawback of this proposed architecture for basic abstractions is that the reduction network for sets with large cardinalities can become extremely deep and untrainable due to a phenomenon called delta attenuation, where the error terms become arbitrarily close to zero and weights never get changed. It is important that we consider other alternatives.

Figure 2.4 shows an alternative architecture with a flat hierarchy. A reduction network is constructed for each of the constituents of a basic abstraction. In the case of set, this

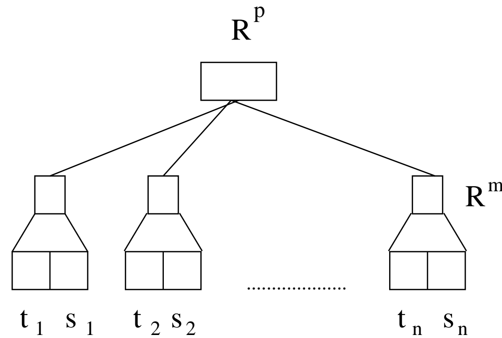


Figure 2.4: Abstraction reduction network - alternative 2

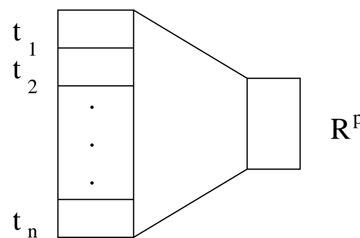


Figure 2.5: Tuple reduction network

corresponds to the elements of the set. These reduction networks are all exactly identical, including their weights. Depending on the type of the constituents, the reduction networks may take the form of structure reduction network, tuple reduction network or even another abstraction encoding network. The outputs of all the individual constituents are connected with the same weight to a top-most processing unit. This architecture is similar to the multi-instance neural networks of [RR00]. Other feasible variants of this architecture can be constructed.

The design of the reduction network for basic tuples is almost trivial, and is illustrated in Figure 2.5. Its proper functioning, however, rests on the premise that its constituent elements can be properly reduced to real-valued vectors. It is worth noting that in the case of tuples of constants, our tuple reduction network degenerates into a conventional $\mathbb{R}^n$ to $\mathbb{R}^p$ neural network.

Each rectangle representing a term in Figures 2.2 to 2.5 may consists of a variable number of neurons, the only constraint being that rectangles representing terms of the same type must have the same number of neurons. This condition is necessary and sufficient to ensure that all instances of reduction networks can be properly connected. The exact number of neurons to put into any particular rectangle is a decision to be made at design time. It is usually a function of the relative complexity of the term the network is intended

to reduce. The internal architectures of reduction networks are also unspecified. It turns out that the design of the individual network topologies has a great impact on the learnability of term reduction networks. This very interesting topic will be further explored in Section 2.3.1.

The networks shown in Figures 2.2, 2.3 and 2.5 were first introduced by J.W. Lloyd in [Llo00]. They were named structure encoding network, abstraction encoding network and tuple encoding network respectively in that paper, reflecting their roles in an earlier architecture. In this thesis, we have renamed them to reflect the current view.

Putting It All Together

For a problem domain where the individuals are all basic terms of type α, where $\alpha \in \mathfrak{S}^c$, the set of basic network components needed to construct all possible individuals must be constructed. To that end, we need to collect together the types of all subterms of terms in $\mathfrak{B}_\alpha$. The set *embed(α)* of *embedded types* in α is defined as follows [Llo01]:

Definition. The set $embed(\alpha)$ of *embedded types* in α is defined by $embed(\alpha) = \{\beta \mid$ there exists $t \in \mathfrak{B}_\alpha$ and a subterm s of t such that $s \in \mathfrak{B}_\beta\}$.

A network component is constructed for each non-atomic type in *embed(α)* as follows: For every type in *embed(α)* of the form $T\alpha_1 \dots \alpha_k$ and every data constructor C having signature $\sigma_1 \to \dots \to \sigma_n \to (Ta_1 \dots a_k), n > 0$ such that there is a type substitution γ with $(Ta_1 \dots a_k)\gamma = T\alpha_1 \dots \alpha_k$, there is a structure reduction network for C. For every type in *embed(α)* having the form $\beta \to \gamma$, there is an abstraction reduction network. For every type in *embed(α)* having the form $\alpha_1 \times \dots \times \alpha_n$, there is a tuple reduction network. This procedure was first introduced in [Llo00].

We call the set of all reduction networks so constructed the set of *component networks*. Together, the component networks make up the complete set of basic building blocks necessary to construct neural networks for all basic terms of type α. Networks that are formed using compositions of instances of component networks are called *composite networks*. A complete composite network constructed from a basic term of type α, which represents an indvidual in the application domain, is called an *individual network*.

An Illustrative Example

We end this section with an example that solidifies the ideas presented so far.

Suppose every individual in an application can be described by three separate components, a binary tree whose nodes take on $Char$ values, a set of integers and a list of real numbers. The individuals have type $BTree\ Char \times \{Int\} \times List\ Float$. Suppose also that there are two data constructors, $BNode$ of arity 3 having the signature $BTree\ Char \to Char \to BTree\ Char \to BTree\ Char$ and $Null$ of arity 0 having the signature $BTree\ Char$. Let $\alpha = BTree\ Char \times \{Int\} \times List\ Float$, then

$$embed(\alpha) = \{BTree\ Char \times \{Int\} \times List\ Float, BTree\ Char, \\ \{Int\}, List\ Float, Char, Int, \Omega, Float\}.$$

Thus, the reduction networks that will be needed are as follows:

1. A tuple reduction network corresponding to $BTree\ Char \times \{Int\} \times List\ Float$.
2. A structure reduction network corresponding to $BNode : BTree\ Char \rightarrow Char \rightarrow BTree\ Char \rightarrow BTree\ Char$.
3. An abstraction reduction network corresponding to $\{Int\}$.
4. A structure reduction network corresponding to $cons : Float \rightarrow List\ Float \rightarrow List\ Float$.

Reduction for nullary data constructors like $Null$, characters, integers and floating point numbers are conducted directly using a static labelling function.

Armed with the reduction networks and the labelling function, we can now construct neural networks for any term of type α. As an example, consider the following basic term denoting an individual of type α.

$$t = (BNode(BNode\ Null\ 'A'\ Null)\ 'B'\ (BNode\ Null\ 'C'\ Null), \{3, 10, 15\}, [20.0, 14.0])$$

Figure 2.6 shows the neural network representation of t. It is constructed in a bottom up fashion. Note that it contains replicated instances of the component networks. For example, there are 3 $\{Int\}$ abstraction reduction networks and two *cons* structure reduction networks.

The reader may notice at this stage that different terms denoting individuals in the application domain can lead to rather different looking individual networks. If so, how can learning be conducted when the different individual networks can have different sets of weights? The short answer is that it is the weights of the component networks, and not those of the individual networks, that are learnt during the training process. Chapter 3 deals with this in more detail.

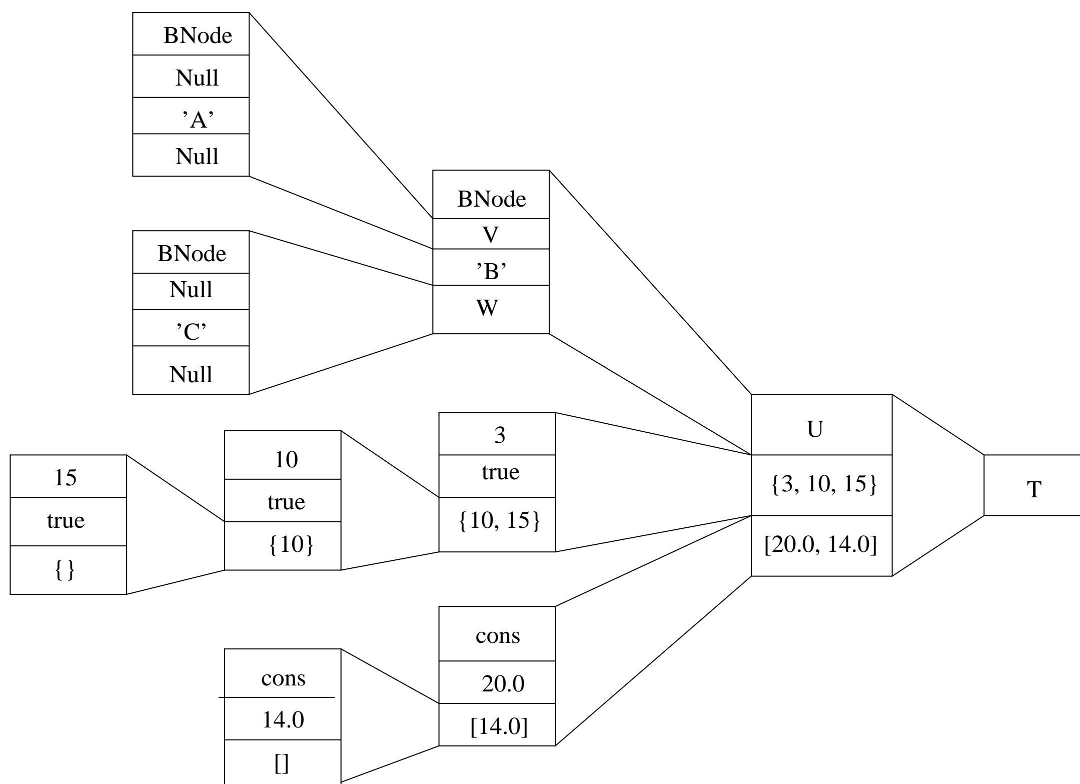
2.3 Design Issues for Term Reduction Networks

We conclude with a discussion of some important design issues for term reduction networks.

2.3.1 Network Topology

The exact internal architectures of reduction networks are unspecified. The topology can take on different shapes. The network processing units can be linear or non-linear units, or a combination of the two. In the case of non-linear units, there is a further decision to be made about the exact form of the non-linear function to be used. The logistic and tanh functions are popular choices, but other alternatives are possible. Appendix D of [RM99] lists several feasible functions.

Expert background knowledge, if available, can be used to decide on the best topology for each component. In the case when no such knowledge is readily available, a default



T = (BNode (BNode Null 'A' Null) 'B' (BNode Null 'C' Null), {3, 10, 15}, [20.0, 14.0])
U = (BNode (BNode Null 'A' Null) 'B' (BNode Null 'C' Null)
V = BNode Null 'A' Null
W = BNode Null 'C' Null

Figure 2.6: Neural network for the term (BNode (BNode Null 'A' Null) 'B' (BNode Null 'C' Null), $\{3, 10, 15\}$, [20.0, 14.0])

configuration of say, one hidden layer of n nodes can be used. Trial and error is another way to discover good topologies. Several more systematic strategies for doing architecture selection can be found in [Moo94]. It is also possible to employ adaptive neural network techniques like the cascade correlation architecture [FL90] or other forms of evolving neural networks (see [BH95] and [Yao99] for two excellent surveys) to learn the internal architectures dynamically. In fact, recursive neural networks for structures, a class of recurrent networks for handling *labelled directed ordered acyclic graphs*, uses the cascade correlation technique to develop the internal architectures adaptively. See [BMSS00] for details.

The exact topologies of choice for the component networks in an application can have huge ramifications on the learnability of the final individual networks. The topology of a component network determines the class of functions that the component can approximate. The class of functions in turn determines the type of transformation that is performed on

the input vector. There are many different types of functions. I identify two general classes to which they belong, based on their behaviour in the overall network. The first class is the so-called reduced representation generator. The ability of neural networks to learn compact representations of the original input using some form of automated feature selection/extraction/construction mechanism is well-documented. Networks of this kind implement functions that compute reduced descriptions of the input in the manner discussed in [Hin90a]. The second class is what I call the relational processor. Networks in this class treat the individual inputs as entities and compute relationships between them. Networks that simulate classification or regression functions, for example, belong to this class. So do logical networks that implement boolean functions. This class of networks is not unlike knowledge-based artificial neural networks (KBANN), which are built on Horn clauses [TS93, Sha92]. Viewed this way, an individual network composed of different components can be thought of as a semantic network, with some component instances acting as nodes representing objects and concepts, and others representing the relationships between them, as defined by the functions learned.

The process by which the behaviour of a particular component network is determined is largely dynamic. It is part of the learning process. In certain circumstances, however, it maybe necessary to statically force a particular type of network to assume a specific role. This can be done at compile time using specialised architectures and fixed weights. An example of this can be found in Section 4.2.2.

2.3.2 The Labelling Function

An important consideration in applying term reduction networks is the design of the input encoding scheme. In principle, neural networks can perform arbitrary non-linear functional transformations, and the exact forms of the inputs should not matter. In practice, however, neural networks can be very sensitive to ill-processed inputs. Inputs that are too large can cause network saturation early on in the learning process; inputs that are too small can result in long training times because larger weights are needed to distinguish between them; inputs that contain values of differing scales can cause problems during training if not regularised properly; etc. There are different ways to properly encode input values, depending on whether it is categorical or continuous in nature.

Encoding Categorical Values

A common way to encode categorical values is the 1 of C encoding scheme, where C is the number of constants. Each constant is represented using a C-tuple, with 0's for all but one of the elements. For example, consider the three categorical values *Red*, *Blue* and *Green* of type *Colour*. Using 1 of C encoding, the three will be represented as (0,0,1), (0,1,0) and (1,0,0) respectively.

Categorical values should *not* be encoded using a single variable. For example, we should not encode the three colours above using the values 0.1, 0.5 and 0.9 because the order artificially introduced into the encoded values can have undesirable effects on the

function the network learns. Readers are referred to Part 2 of the Neural Network FAQ [Sar00] for a more detail discussion on this topic.

Encoding Continuous Values

An element represented as a continuous value has a type which determines its range of admissible values. For an example, let us consider the following three bits of information that make up part of the input to a network: a person's age, her annual income, and the probability that she will get a pay rise this year. The range of the age element is 0 to 150; the range of the annual income element is $35,000 to $500,000; and the range of the probability element is 0 to 1. Note that typical values for each of the three elements can differ from each other by up to a few orders of magnitude, but the size of the elements usually have no direct relationship with the importance of the element as a discriminating feature. If we were to naively input them to the network without any preprocessing, at best, learning will be a hard and lengthy process; at worst, learning cannot proceed.

There are two ways to encode continuous values properly. The first is to normalise the inputs by applying a linear transformation on them. For each element x, we compute its mean $\overline{x}$ and variance σ^2 with respect to the training data, and then rescale all the instances x_i for $1 \leq i \leq N$ of the element using the formula

$$\widetilde{x_i} = \frac{x_n - \overline{x}}{\sigma}$$

to arrive at a set of similarly-valued values. This simple normalisation procedure makes the assumption that the elements are independent variables. This is of course not always true. There are more sophisticated techniques that can take into account correlations among the elements. Readers are referred to [Bis95, sec 8.2] for more details on that.

The second acceptable encoding scheme is to simply discretise the continuous values, partitioning them into intervals, and use the interval labels as inputs. Presumably, the interval labels for each element are also continuous in nature, but their values will usually have, or can be designed to have, the same order of magnitude. There are different techniques that can be used to find good partitions. These include binning, histogram analysis, cluster analysis, entropy-based discretisation, and segmentation by natural partitioning. The exact details of how each of them works are beyond the scope of this thesis. For a good survey of the techniques, see [HK01, sec 3.5].

Chapter Notes

The simple theory of types, the logic upon which the formal knowledge representation language used in this thesis is based, was developed by the celebrated American logician Alonzo Church in 1940. Church developed lambda calculus, with the help of his student Stephen C. Kleene in the 1930's in his study of computability. He later incorporated types into the lambda calculus, the result of which is the simple theory of types. It is more commonly referred to as the simply typed lambda calculus. For accessible accounts of simply typed lambda calculus, see [Mit96], [And86] and [Wol93]. For a lay-person guide to lambda calculus, see [Pen89]. For an interesting account of the theory of logical types, see [Cop71].

Logic doesn't apply to the real world.
Marvin Lee Minsky

Chapter 3

Computation

What is it that gives us the feeling of elegance in a solution, in a demonstration?
It is the harmony of the diverse parts, their symmetry, their happy balance;
in a word it is all that introduces order, all that gives unity,
that permits us to see clearly and to comprehend at once
both the ensemble and the details.
Jules Henri Poincaré

Computation in neural network learning deals with the problem of how the system should best make use of the training data available to compute its predictions. This very important issue is discussed in this chapter. We begin the investigation by formulating the task as a function optimisation problem. We avail ourselves of the extensive works on function optimisation, a very well-researched area in numerical methods, and explore possible approaches to solving our computation task. We then propose in Section 3.2 a simple and elegant solution based on gradient descent in the form of the Back-propagation Through Structure algorithm. We also trace the algorithm to its origin in works on folding architectures, and point out the modifications needed to make it work for term reduction networks.

3.1 An Exercise in Function Optimisation

A neural network is essentially a function of the form $f(x, w)$, where x is the input individual and w the weights of the network. The return value of $f(x, w)$ is a real-valued vector. Let us denote the output of the function for individual x_i by o_i. Corresponding to each individual x_i is a target value t_i, which is also a real-valued vector having the same dimension as o_i. The aim of a network training algorithm (or computational method, if termination is not guaranteed) is to, as much as possible, make the values of o_i and t_i as close as possible for all i, by changing the values of w. Denoting the dissimilarity between o_i and t_i by a function $D(t_i, f(x_i, w))$, and summing over all i to get $E = \sum_i D(t_i, f(x_i, w))$, we effectively obtain a measure of how good a network is doing its job at predicting target

values for the input individuals. For consistency with literature, let us call this measure the error function of the network. Now, computation is really about utilising the training data to find a value of w which will minimise this function. That is, computation is really a function minimisation problem. The dimension of w determines whether we have a single variable or multivariate function minimisation problem. Likewise, the range of w determines whether we have a constrained or unconstrained optimisation task. Different methods exists for each of the four possible combinations. [PTVF92, chap 10] and [Hea96, chap 6] contain excellent discussions of efficient algorithms for each of the four classes of optimisation problems.

We have formulated the above as a minimisation problem. We could as well have posed it as a maximisation problem, by defining a similarity (as opposed to dissimilarity) measure between o_i and t_i and try to maximise that for all i. This formulation is however relatively uncommon.

3.1.1 The Error Function

The choice of the error function is dependent on the exact nature of the learning task. For regression problems, the goal is to model the conditional distribution of the output variables, conditioned on the input variables [Bis95]. For this purpose, the sum of squared errors (SSE) function is a good choice. SSE is defined as

$$E_{SSE} = \frac{1}{2}\sum_{i}^{N}\sum_{j}^{M}(t_{ji} - o_{ji})^2$$

where N is the total number of individuals, M is the dimension of the output vector, *i.e.*, the number of units in the output layer, and t_{ji} (resp. o_{ji}) denotes the target (resp. output) value of unit i for individual j.

For classification problems, the goal is to model the posterior probabilities of class membership, again conditioned on the input variables [Bis95]. For this purpose, the SSE function is still applicable, but there exist other more appropriate functions. One popular choice is the logarithmic or cross-entropy error function, defined as

$$E_{log} = \sum_{i}^{N}\sum_{j}^{M}(t_{ji}\ln o_{ji} + (1 - t_{ji})\ln(1 - o_{ji}))$$

where o_{ji} in this case is interpreted as the estimated probability that individual j belongs to class i and $t_{ji} \in \{0, 1\}$ is the target value.

Different error functions carry with them different assumptions about the model and distribution of fitting errors. For example, the SSE function corresponds to a maximum likelihood model with the assumption that errors have a Gaussian distribution, whereas the cross-entropy error function corresponds to a classification model and the assumption of a binomial error distribution [RM99]. In practice, an incorrect choice may lead to poor

generalisation performance if the underlying assumptions of the error function are not met by the model and data available.

Readers are referred to [Bis95, chap 6] for a detail and extensive coverage of error functions and their properties. In the remainder of this chapter, we shall assume the use of the SSE function for all our learning tasks. This preference for SSE is primarily motivated by analytical simplicity.

3.1.2 Optimisation Techniques

Having looked at possible forms of objective functions, we now turn to optimisation techniques. Function optimisation is an old discipline in numerical methods. A lot of rather wonderful methods have been proposed in the past. Methods as old as Newton and as young as Brent are in wide popular use. These can be conveniently classified into two categories: those that make use of the gradient information and those that do not. Some methods guarantee the location of the global optimum, while others can only find a local optimum. Naturally, the former usually require much more computational resources than the latter. There is no one single best technique for all scenarios. The most appropriate method for a particular problem is largely dependent on the properties of the objective function, and the constraints (if any) placed on the possible solutions.

Error back-propagation, an algorithm based on gradient descent, is one of the most important and popular optimisation techniques in neural network learning. It is important for historical reasons, in fact, its discovery in the eighties is widely regarded as one of the most important factors behind the revival of the field of neural networks research following a decade long slump instigated by the publication of [MP69]. The algorithm's popularity is a result of its remarkable simplicity and the elegance of the mathematics behind it.

Computation in neural network learning is an active field of research, perhaps *the* most active field in neural networks research. Various algorithms which are better than standard error back-propagation in one way or another have been put forward in the last decade or so. On the performance side, a lot of efforts have been made to come up with faster variations of error back-propagation. These includes the Delta-Bar-Delta method [Jac88], Vogl's method [VMR$^+$88], Rprop [RB93], Quickprop [Fah89], SuperSAB [Tol90], the Silva and Almeida method [SA90], etc. Besides these variants of standard back-propagation, alternative algorithms, most of which are based on classical optimisation techniques have also been proposed in the literature. The most widely used alternative is probably the conjugate gradient descent method. Interested readers are referred to [RM99, chap 9-10] and [Bis95, chap 7] for comprehensive treatments on these methods.

For the purpose of training term reduction networks, most if not all of the techniques mentioned above, with appropriate modifications, can be used. In this thesis, we extend as an exercise an algorithm called Back-propagation Through Structure based upon the standard error back-propagation algorithm. It was introduced by C. Goller and A. Küchler for their folding architecture [GK96], a class of recurrent neural networks not unlike our structure reduction networks. This is discussed in the next section. The aim is to point out some of the special requirements of term reduction networks as a result their slightly

strange (but beautiful!) architectures, and ways to satisfy those requirements. Other more advanced training methods can be similarly extended for term reduction networks.

3.2 Error Back-Propagation

3.2.1 Standard Error Back-Propagation

We begin with a brief review of the standard error back-propagation (SEBP) algorithm. Readers with background in neural network learning may wish to skip this part and go straight to section 3.2.2.

The intuition behind SEBP is simple. Given a set of training examples, the SEBP algorithm works by iteratively making adjustments to the network weights in such a way that the error function of the network with respect to the training set is minimised. To do that, the error term is repeatedly propagated through the network from the output units to the input units to calculate the contribution of each weight to the error term. Adjustments are then made to the weight based on their "badness". This backward motion of the error term through the network is the rationale behind the name 'error back-propagation'. The network weights are usually initialised to some small random numbers in the beginning.

Error back-propagation consists of two distinct phases: the first phase computes the (partial) differential of the error function with respect to each weight in the network; the second phase updates the weights according to a weight update rule. In the standard back-propagation algorithm, the gradient is computed for each individual in the training set and then summed together to arrive at the final gradient. Weights are updated once every epoch, where an epoch is a complete processing cycle involving every individual in the training set. In practice, this is extremely slow. Commonly, a stochastic approximation of SEBP is used. Under this scheme, weights are updated after the gradient is calculated for each individual.

Before giving the exact steps of the algorithm, it will be helpful at this stage to introduce a consistent set of notations. In the remainder of this text, we have:

- x_{ji} = the input from unit i to unit j
- w_{ji} = the weight for the link from unit i to unit j
- $net_j = \sum_i w_{ji} x_{ji}$ (the weighted sum of inputs for unit j)
- o_j = the output for unit j
- t_j = the target output for unit j
- $outputs$ = the set of units in the final output layer
- $next(j)$ = the set of units whose immediate inputs include the output of unit j
- σ = the sigmoid function defined by $\sigma(x) = 1/\left(1 + e^{-x}\right)$.

Algorithm SSEBP (Stochastic Error Back-propagation algorithm) Given a neural network NN with uninitialised weights, a set of training examples $TS = \{(d_i, t_i)\}$, and a learning rate η, find a weight solution that (locally) minimises the error function.

S1. [Initialise the weights] Initialise all the network weights to some random floating point number in a certain interval, *e.g.,* between -0.5 and 0.5.

S2. [Train the network] For each $(d_n, t_n) \in TS$ do

1. [Propagate the input forward] Input individual d_n into the network and compute the values at the output units.

2. [Propagate the error backward]

i [Compute gradient] For each network weight w_{ji} from the output units to the input units compute the gradient $\frac{\partial E_{d_n}}{\partial w_{ji}}$.
For output units, $\frac{\partial E_{d_n}}{\partial w_{ji}} = -(t_j - o_j)o_j(1 - o_j)x_{ji}$.
For hidden units, $\frac{\partial E_{d_n}}{\partial w_{ji}} = -o_j(1 - o_j)(\sum_{k \in next(j)} w_{kj}(-\frac{\partial E_{d_n}}{\partial net_k}))x_{ji}$

ii [Update weights] Update each network weight w_{ij} with the rule $w_{ij} \leftarrow w_{ij} - \eta \frac{\partial E_{x_i}}{\partial w_{ij}}$.

S3. [Check termination condition] If the termination condition is met, terminate with success. Otherwise go back to **S2**.

Figure 3.1: Stochastic gradient descent version of standard error back-propagation.

Figure 3.1 gives the stochastic gradient descent version of the standard error back-propagation algorithm.

Step S1 is probably the least understood step of the algorithm. The difference between a good and a bad initial weight is, at best, the difference between a fast and a slow training time, and, at worse, the difference between convergence and non-convergence (within a reasonable amount of time). This (almost extreme) sensitivity of back-propagation to the initial weights was vividly demonstrated in a large number of experiments reported in [KP91]. Various methods have been suggested on how best to initialise a network, but they remain, rules of thumb. A good study on this subject can be found in [TF97]. In it, for networks with non-linear units, the authors suggested the use of a random number in the interval $[-a/\sqrt{d_{in}}, a/\sqrt{d_{in}}]$, where a is a parameter, and d_{in} the fan-in of the neuron to which the weight is connected to, for obtaining good initial weights. (This method is attributed to L.-Y. Bottou by Thimm and Fiesler.) This is the weight initialisation procedure of choice for our experiments in Chapter 4. The weight initialisation step, of course, need not be a random process. Prior structural and domain knowledge about the target function can be used to (partially) initialise a network. More information on

non-random weight initialisation can be found in [RM99, sec 7.2].

The main component of the algorithm is step S2. After the forward phase, we obtain the output for each processing unit in the network, o_i for unit i. From that and the target values t_i for output unit i, the gradient $\frac{\partial E_{d_n}}{\partial w}$ can be calculated. Note that the error term used here is the error for individual d_n, not for the whole training set. The gradient points in the direction of steepest ascent on the error surface. On each iteration, the weight update rule moves the solution in the opposite direction of the gradient, *i.e.,* the direction of steepest descent. The learning rate η defines the step-size of this motion. This is usually a value between 0.05 and 0.5 [RM99]. Small values may result in slow learning, whereas large values may cause the network to overshoot and miss good solution points.

The formula for calculating the gradient component for weight w_{ji} is different for weights connecting to output and to hidden units. The derivatives for weights connecting to output units can be calculated directly. The calculation of the derivatives for weights connecting to hidden units make use of the delta terms ($-\frac{\partial E}{\partial net_k}$ for unit k) for units to which they connect. This dependency mandates the backward motion of the error propagation. Appendix A gives a standard derivation of the gradient formulae and the weight update rule.

The termination condition is another grey area in SEBP. A good termination condition ensures that the network is not over-trained or under-trained, both of which can result in poor generalisation performance. What is certain is that the termination condition must be stated in such a way that the learning process stops after a finite number of steps. This is important, because SEBP is an algorithm, and hence it must satisfy one of the most important properties of an algorithm, finiteness. (See section 1.1 of [Knu97] for the other four properties that a computational method must satisfy to pass the algorithm qualification test.)

Figure 3.2 shows the error surface for a simple network with a single adjustable weight. Using this simple diagram, we can visualise how SEBP works in practice. The weight initialisation step places the network on a random point along the curve. From then on, the algorithm repeatedly calculates the gradient and moves the network a step closer to a solution. For instance, starting from position S1, the network will end up on global minimum A after a finite number of steps. Alternatively, if the network had started from S2, it would have landed on the local minimum B. This simple example illustrates two important points:

1. The SEBP algorithm does not always converge to the global minimum. On error surfaces with many local minima, back-propagation can be trapped at a bad local minimum.
2. The initial starting point can have huge ramifications on the ability of the network to converge to a good solution.

Some heuristics to overcome these difficulties and to make back-propagation perform better can be found in [Hay99, sec 4.6].

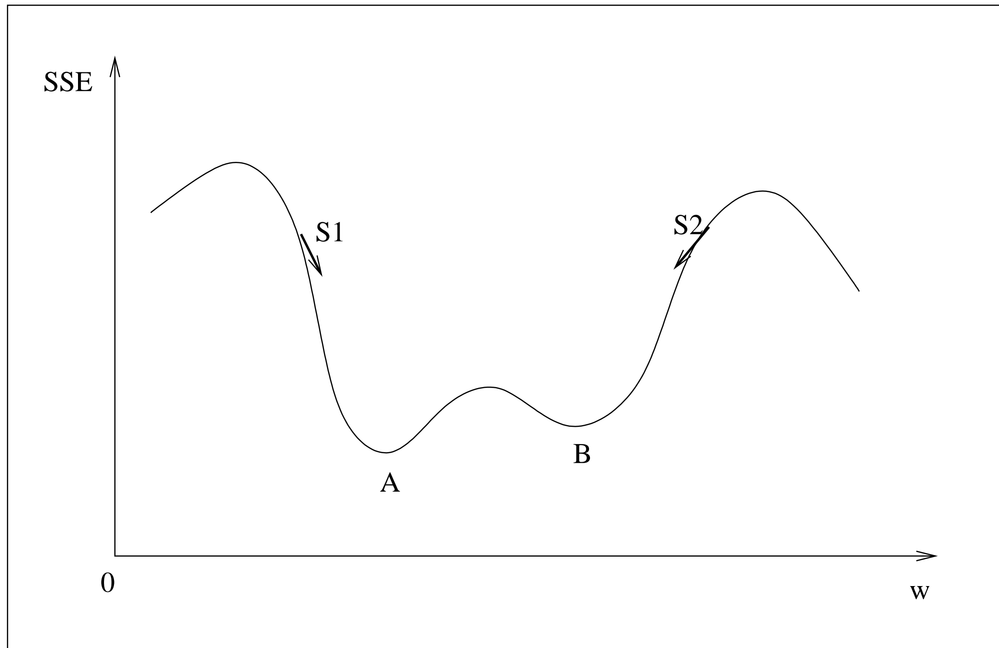


Figure 3.2: The error surface of a simple network with one adjustable weight.

3.2.2 Back-Propagation Through Structure

With some extensions, the standard error back-propagation can be used to train term reduction networks. In this section, we investigate these extensions in detail.

Consider an application where the individuals have type α. Corresponding to the non-atomic embedded types of α are component reduction networks that constitute the building blocks of the individual networks. We have seen in Section 2.2 that multiple instances of the same component network can be used in the construction of an individual network. These different instances are just copies of the original component network, and their weights must therefore stay the same at all times. But error back-propagation will violate this constraint when run in normal mode, updating each copy of the same weight in a different way. The only extension to SEBP we need for term reduction networks is a mechanism to ensure this does not happen.

The point to bear in mind is that our objective is to learn the weights of the component networks that make up the individual networks, from their occurences in the individual networks. Let us denote the weight from unit i to unit j in a component network by w_{ji}, and the same weight in the mth instance of the component network in the individual network by $w_{ji}^{(m)}$. There are two ways to formulate the error function, one with respect to the weights in the component networks, and the other with respect to the weights in the individual networks. We denote the former as E, and the latter as E'. E is a function of

w_{ji} for all i and j. E' is a function of $w_{ji}^{(m)}$ for all i, j and m. The two are connected by the relations

$$w_{ji} = w_{ji}^{(m)}, \ \forall\, i, j, m.$$

We are interested in the values of $\frac{\partial E}{\partial w_{ji}}$, but these cannot be computed directly from the individual networks. What we do have, from running standard error back-propagation through the individual networks, are the values of $\frac{\partial E'}{\partial w_{ji}^{(m)}}$. What is the relationship between the two? We conjectured that the partial differential of E with respect to the weight w_{ji} is simply the sum of all the partial differentials of E' with respect to copies of the weight w_{ji} in the individual network. This we now prove. (This proof is due to J.W. Lloyd [Llo00] based on C. Goller's ideas, see [Gol97, p.48-49].)

Proposition 3.2.2.1. *Let $f : \mathbb{R}^n \to \mathbb{R}$ be continuously differentiable and $1 \leq m < n$. Let $g : \mathbb{R}^m \to \mathbb{R}^n$ be defined by $g(x_1, \ldots, x_m) = (x_1, \ldots, x_s, x_q, x_{s+1}, \ldots, x_t, x_q, x_{t+1}, \ldots, x_m)$, that is, g inserts $(n-m)$ copies of x_q $(1 \leq q \leq m)$ into $(x_1, \ldots, x_m)$ so that x_q ends up at the $(n-m)+1$ positions $i_1, \ldots, i_r$ in $(x_1, \ldots, x_s, x_q, x_{s+1}, \ldots, x_t, x_q, x_{t+1}, \ldots, x_m)$. Then*

$$\frac{\partial (f \circ g)}{\partial x_q} = \frac{\partial f}{\partial y_{i_1}} + \cdots + \frac{\partial f}{\partial y_{i_r}}.$$

Proof: Let $g(x_1, \ldots, x_m) = (y_1, \ldots, y_n)$. Then, by the chain rule,

$$\begin{aligned} &\frac{\partial (f \circ g)}{\partial x_q} \\ =&\frac{\partial f}{\partial y_1}\frac{\partial y_1}{\partial x_q} + \cdots + \frac{\partial f}{\partial y_n}\frac{\partial y_n}{\partial x_q} \\ =&\frac{\partial f}{\partial y_{i_1}} + \cdots + \frac{\partial f}{\partial y_{i_r}}. \end{aligned}$$

□

Proposition 3.2.2.2. $\frac{\partial E}{\partial w_{ij}} = \sum_m \frac{\partial E'}{\partial w_{ij}^{(m)}}$.

Proof: The result follows immediately from Proposition 3.2.2.1. E' is f and E is obtained from $E' \circ g$ where g identifies all the input variables $w_{ij}^{(m)}$ with w_{ij}.

□

A similar technique as that described above was used by C. Goller and A. Küchler to solve a problem of identical nature for folding architectures, a class of recurrent neural networks that when unfolded, looks exactly like our structure reduction network. (See [GK96], [KG96] and [Gol97]). They named their algorithm back-propagation through structure (BPTS), in analogy with the back-propagation through time algorithm. We shall use the name back-propagation through structure for our algorithm as well, considering the similarity between the two.

Algorithm SBPTS (Stochastic Back-Propagation Through Structure algorithm) Given the set of training examples $D = \{\langle d_i, t_i \rangle\}$, the type of the individuals α, and the learning rate η, construct the set of network components necessary to build reduction networks for the individuals, then find and return a weight solution for the network components such that the error of the individual networks composed of these components with respect to the training examples is (locally) minimised.

S1. [Initialise the data structures] Initialise the set of individual networks *individuals* and the set of network components *components* to the empty set. $individuals \leftarrow \phi$. $components \leftarrow \phi$. Initialise the set *embeddedTypes* to the embedded types of α. $embeddedTypes \leftarrow embed(\alpha)$.

S2. [Build the network components] For each non-atomic type $\gamma \in$ *embeddedTypes*, construct a component network for it, initialise its weights using random floating point numbers in a certain interval and then add the component to the set *components*.

S3. [Initialise the weight derivatives matrix] Initialise a 2 dimensional $N \times N$ matrix $derivatives[N][N]$ to zero, where N is the total number of processing units in all of the network components. This matrix is needed to compute $\frac{\partial E_d}{\partial w}$ from $\frac{\partial E'_d}{\partial w^{(m)}}$.

S4. [Find a better weight solution] For each $\langle d_k, t_k \rangle \in D$ do

1. [Construct individual network] Construct an individual network NN for d_k using the network components.
2. [Propagate the input forward] Compute the activation energy at each processing units, from the input layers to the output layer.
3. [Propagate the error backward]
   - **i** [Calculate $\frac{\partial E'_{d_k}}{\partial w^{(m)}}$ and update *derivatives*] For each $w_{ji}^{(m)}$ in NN from the output units to the input units compute the gradient $\frac{\partial E'_{d_k}}{\partial w_{ji}^{(m)}}$ and then set $derivatives[i][j] \leftarrow derivatives[i][j] + \frac{\partial E'_{d_k}}{\partial w_{ji}^{(m)}}$
   - **ii** [Update component weights] For each w_{ji} in the network components set $w_{ji} \leftarrow w_{ji} + \eta \cdot derivatives[i][j]$
4. [Reinitialise *derivatives*] Reset the matrix *derivatives* to zero.
5. [Destroy the individual network NN] Free the memory occupied by NN.

S5. [Check termination condition] If termination condition is met, terminate with success. Otherwise, go back to step S4.

Figure 3.3: The stochastic gradient descent version of Back-propagation through Structure.

We can now give the exact steps of the algorithm. Figure 3.3 gives the stochastic gradient descent version of the back-propagation through structure algorithm.

At step S2, the user may be prompted for information on the exact architecture of each component. Information required may involve the number of units to encode each type, the number of hidden layers in a component network, the number of units in each hidden layer, etc. Alternatively, a default architecture may be used. At step S4, the gradients are calculated in the usual way using the formulae given in Algorithm SSEBP (Figure 3.1).

Note that the individual networks are constructed and destroyed on each iteration. This is done primarily to achieve clarity in presentation, but is also intended to illustrate a potential problem with term reduction networks and BPTS: inefficient memory usage. For small problems, the individual networks should be constructed once in an initialisation step. But for larger problems with a training set of reasonably big size and/or individuals of highly non-trivial complexity, it may not be possible to fit all the individual networks in memory at all times. In such cases, dynamic creation and destruction of individual networks may be necessary.

I conclude the discussion of our network training algorithm with a brief discussion on an alternative weight update formula. It has the same general form

$$w_{ji}^{t+1} = w_{ji}^{t} + \triangle w_{ji}^{t} \tag{3.1}$$

where w_{ji}^{t} denotes the value of w_{ji} at time t. The update term is defined as follows,

$$\triangle w_{ji}^{t} = -\eta \frac{\partial E}{\partial w_{ji}^{t}} + \mu \triangle w_{ji}^{t-1} \tag{3.2}$$

with an additional *momentum* term. Note that the weight update term is now a function of all previous update terms. The addition of the momentum term effectively introduces inertia to the motion through the weight space, resulting in smoother trajectories. This usually leads to faster convergence in training time. It is this weight update rule that we use in our experiments in the next chapter.

Chapter 4

Results

Result! Why, man, I have gotten a lot of results.
I know several thousand things that won't work.
Thomas Edison

In this chapter, I present the results of applying term reduction networks to eight classification problems.

4.1 Five Easy Pieces

This section presents the result of applying term reduction networks to five (arguably) trivial problems. The accuracies obtained for these problems are not statistically meaningful, due to the large discrepancies between the degrees of freedom in the networks used to solve them and the small number of training examples available. The purpose of the exercise is only to illustrate the diverse range of problems that our networks are capable of solving.

4.1.1 Tennis (Tuple)

This is the classic tennis problem proposed by Quinlan [Qui86], [Mit97, p.59]. It is an attribute-value language problem, and involves learning the concept of whether to play tennis given the weather on a Saturday.

Problem Specification We introduce the types *Outlook*, *Temperature*, *Humidity*, *Wind* and a type synonym *Weather* as follows.

$$Sunny, Overcast, Rain : Outlook$$
$$Hot, Mild, Cool : Temperature$$
$$High, Normal, Low : Humidity$$
$$Strong, Medium, Weak : Wind.$$
$$Weather = Outlook \times Temperature \times Humidity \times Wind.$$

The task is to learn a function with the following signature: $playTennis : Weather \rightarrow \Omega$. The dataset contains nine positive examples and five negative examples as follows:

$$
\begin{aligned}
&playTennis\ (Overcast, Hot, High, Weak) = \top\\
&playTennis\ (Rain, Mild, High, Weak) = \top\\
&playTennis\ (Rain, Cool, Normal, Weak) = \top\\
&playTennis\ (Overcast, Cool, Normal, Strong) = \top\\
&playTennis\ (Sunny, Cool, Normal, Weak) = \top\\
&playTennis\ (Rain, Mild, Normal, Weak) = \top\\
&playTennis\ (Sunny, Mild, Normal, Strong) = \top\\
&playTennis\ (Overcast, Mild, High, Strong) = \top\\
&playTennis\ (Overcast, Hot, Normal, Weak) = \top\\
&playTennis\ (Sunny, Hot, High, Weak) = \bot\\
&playTennis\ (Sunny, Hot, High, Strong) = \bot\\
&playTennis\ (Rain, Cool, Normal, Strong) = \bot\\
&playTennis\ (Sunny, Mild, High, Weak) = \bot\\
&playTennis\ (Rain, Mild, High, Strong) = \bot.
\end{aligned}
$$

Experiment Setting The individuals have type *Weather*, which is just a tuple of constants. Using one hidden layer with two nodes, and a real number to encode the constants, we end up with a network component as shown in Figure 4.1. It is worth noting that in the case of simple tuples of constants, our structured tuple encoding network degenerates into a conventional feedforward neural network. Training is conducted with learning rate set at 0.01 and momentum at 0. The weights are initialised with random values very close to zero.

Experimental Result After about 10,000 epochs, we obtain 100% accuracy on the training set. The final weight solution is shown in Table 4.1. The table can be read as follows: a value in cell (i, j) denotes the weight w_{ij}, the weight from node j to node i. [1]

Discussion Strictly speaking, the labelling function used in this example is inappropriate. Categorical values should really be encoded using 1 of C or 1 of C-1 schemes for the outputs to be validly interpreted as the posterior probability of the individual belonging to the output class. See "How should categories be encoded?" in part two of [Sar00] for further details. But the constants used here have some ordinal properties, in the sense that they can be roughly ranked according to the effect they have on the final target of whether to play tennis. For example, sunny, overcast and rain can be ranked in that order in terms

[1] The weight solutions for this and the following problems are presented for the consumption of a special group of readers interested in rule extraction from neural networks. The average reader may wish to ignore them.

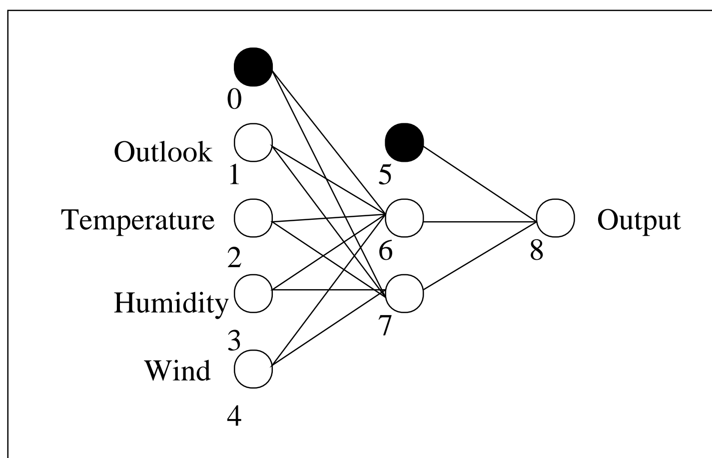


Figure 4.1: The tennis network component. Two nodes (excluding the bias node) were used in the hidden layer. The black nodes are bias nodes with input fixed at 1. All the nodes are numbered for referencing purposes.

Table 4.1: The weight solution for tennis.

i \ j	0	1	2	3	4	5	6	7
6	4.33198	-7.85716	0.605261	0.0529212	5.45755	-	-	-
7	-3.33038	9.99733	2.43514	5.95944	-1.28383	-	-	-
8	-	-	-	-	-	-12.3405	8.43928	8.4348

of their suitability as a weather component in determining whether or not to play tennis. This is the reason behind our choice to encode them as real numbers.

4.1.2 Keys (Set)

Having discussed tuples, we now turn our attention to a problem involving sets. This is the second problem in a series of “Ten Easy Pieces” described in [BGCL01].

Problem Specification The problem involves finding the concept of whether a bunch of keys can open a door. A key is a tuple with four components: make, numprongs, length and width. Each individual in the dataset is a set of keys. The type information is as follows:

$$Abloy, Chubb, Rubo, Yale : Make$$
$$Short, Medium, Long : Length$$
$$Narrow, Normal, Broad : Width.$$

For convenience, we introduce the following type synonyms.

$$\begin{aligned} NumProngs &= Nat \\ Key &= Make \times NumProngs \times Length \times Width \\ Bunch &= \{Key\}. \end{aligned}$$

The task is to learn a function having the following signature: $opens : \{Key\} \rightarrow \Omega$. The dataset contains five positive examples and four negative examples.

$$\begin{aligned}
&opens\ \{(Abloy, 4, Medium, Broad), (Chubb, 3, Long, Narrow), \\
&\qquad (Abloy, 3, Short, Normal)\} = \top \\
&opens\ \{(Abloy, 4, Medium, Broad), (Chubb, 3, Long, Narrow), \\
&\qquad (Abloy, 3, Short, Broad)\} = \top \\
&opens\ \{(Rubo, 5, Short, Narrow), (Yale, 4, Long, Broad), \\
&\qquad (Abloy, 3, Medium, Narrow), (Chubb, 6, Medium, Normal)\} = \top \\
&opens\ \{(Yale, 4, Medium, Broad), (Chubb, 3, Long, Broad), \\
&\qquad (Abloy, 3, Medium, Broad), (Abloy, 4, Medium, Narrow)\} = \top \\
&opens\ \{(Chubb, 4, Medium, Broad), (Chubb, 2, Long, Normal), \\
&\qquad (Abloy, 3, Medium, Broad)\} = \top \\
&opens\ \{(Yale, 3, Short, Narrow), (Yale, 4, Long, Normal), \\
&\qquad (Chubb, 3, Short, Broad), (Chubb, 4, Medium, Broad)\} = \bot \\
&opens\ \{(Yale, 4, Long, Broad), (Yale, 3, Long, Narrow)\} = \bot \\
&opens\ \{(Rubo, 4, Long, Broad), (Yale, 4, Long, Broad), \\
&\qquad (Abloy, 3, Short, Broad), (Chubb, 3, Short, Broad)\} = \bot \\
&opens\ \{(Chubb, 3, Medium, Broad), (Rubo, 5, Long, Narrow), \\
&\qquad (Abloy, 4, Short, Broad)\} = \bot.
\end{aligned}$$

Experiment Setting We use a single hidden layer consisting of two nodes in this experiment. Figure 4.2 shows the basic network components. Learning is conducted with learning rate set at 0.2 and momentum at 0.3. The constants are encoded using real numbers between 0 and 1.

Experimental Result After about 10,000 epochs, the network obtains 100% accuracy on the training data. The weight solution is shown in Table 4.2.

Discussion The original problem setting as proposed by Bowers *et al.* is that of a multiple-instance type problem (see Section 4.2.2 for details of multiple-instance problems). Intuitively, a set of keys can open a door if there exists one key in the set which can open the door. Here, we have relaxed the problem specification and have not imposed the multiple-instance constraint on the final solution. In general, it is difficult (if not impossible) to

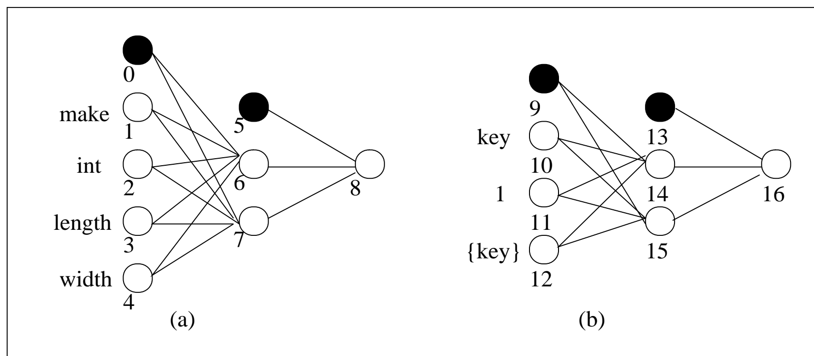


Figure 4.2: The network components. Two nodes (excluding the bias node) were used in the hidden layer. The black nodes are bias nodes with input fixed at 1. (a) The *Key* component. (b) The $\{Key\}$ component.

Table 4.2: The weight solution for the Keys problem.

i \ j	0	1	2	3	4	5	6	7
6	-2.11284	5.85969	-3.2994	-2.66559	5.01864	-	-	-
7	1.1957	-2.22785	-6.4896	-0.396301	1.45484	-	-	-
8	-	-	-	-	-	3.99131	-7.33326	6.48196
i \ j	9	10	11	12	13	14	15	
14	0.55045	8.84594	-0.694437	-5.10838	-	-	-	
15	-1.26107	-0.337535	-1.19349	-2.54679	-	-	-	
16	-	-	-	-	-4.19598	9.14185	1.36091	

capture the precise definition of a symbolic rule using subsymbolic constructs. We will come back to this issue and introduce a way to approximate the multiple-instance constraint in Section 4.2.2 when we deal with the Musk problem. The solution is however somewhat unsatisfactory; for instance, it does not work for this problem.

4.1.3 The East-West Challenge (List)

This section examines a problem involving lists. This problem is the well-known East-West challenge of Michalski [Eas]. It involves learning the concept of whether a train is headed East or West based on properties of its constituent cars. Figure 4.3 shows the set of examples as originally proposed by Michalski [ML77].

Problem Specification We first need to decide the appropriate type for trains. List seems to be the most natural choice here. We introduce the types *Direction*, *Shape*, *Length*,

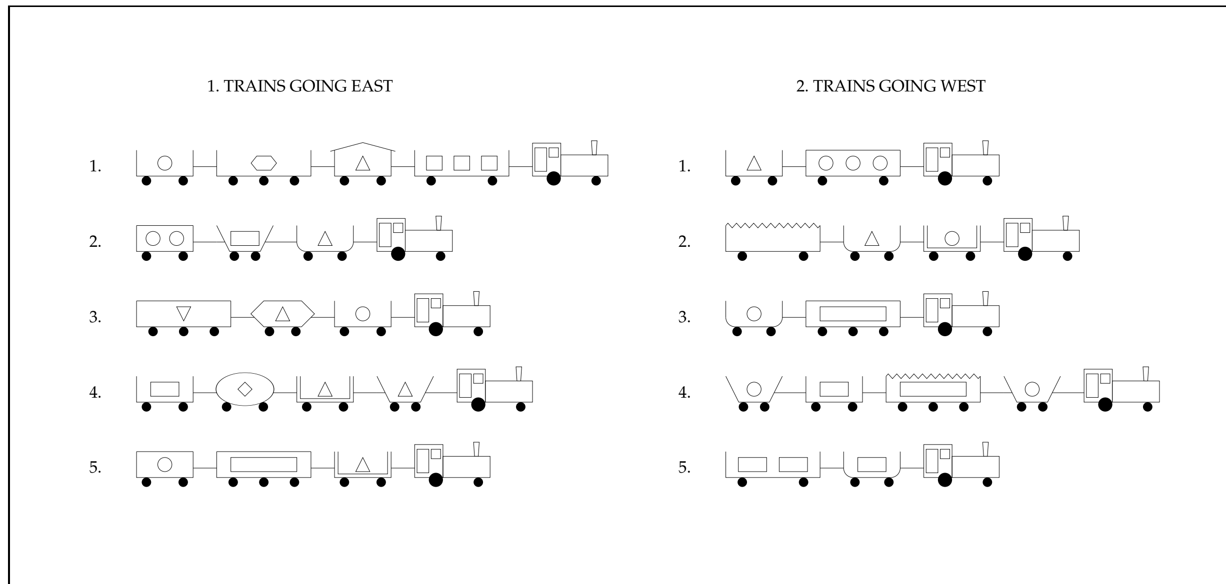


Figure 4.3: Trains in Michalski's East-West Challenge.

Kind, *Roof*, and *Object* as follows:

$$\begin{gathered} East, West : Direction \\ Rectangular, DoubleRectangular, UShaped, BucketShaped, Hexagonal, Ellipsoidal : Shape \\ Long, Short : Length \\ Closed, Open : Kind \\ Flat, Jagged, Peaked, Curved, None : Roof \\ Circle, Hexagon, Square, Rectangle, LongRectangle, Triangle, InvertedTriangle, \\ Diamond, Null : Object. \end{gathered}$$

For convenience, we also introduce the following type synonyms,

$$\begin{gathered} NumWheels = Nat \\ NumObjects = Nat \\ Load = Object \times NumObjects \\ Car = Shape \times Length \times NumWheels \times Kind \times Roof \times Load \\ Train = List\ Car. \end{gathered}$$

We represent each train as a list of cars, and we wish to learn a function with the following signature: $direction : Train \rightarrow Direction$.

The dataset is as follows:

$$
\begin{aligned}
\mathit{direction}\ [&(\mathit{Rectangular}, \mathit{Long}, 2, \mathit{Open}, \mathit{None}, (\mathit{Square}, 3)),\\
&(\mathit{Rectangular}, \mathit{Short}, 2, \mathit{Closed}, \mathit{Peaked}, (\mathit{Triangle}, 1)),\\
&(\mathit{Rectangular}, \mathit{Long}, 3, \mathit{Open}, \mathit{None}, (\mathit{Hexagon}, 1)),\\
&(\mathit{Rectangular}, \mathit{Short}, 2, \mathit{Open}, \mathit{None}, (\mathit{Circle}, 1))] = \mathit{East}\\
\mathit{direction}\ [&(\mathit{UShaped}, \mathit{Short}, 2, \mathit{Open}, \mathit{None}, (\mathit{Triangle}, 1)),\\
&(\mathit{BucketShaped}, \mathit{Short}, 2, \mathit{Open}, \mathit{None}, (\mathit{Rectangle}, 1)),\\
&(\mathit{Rectangular}, \mathit{Short}, 2, \mathit{Closed}, \mathit{Flat}, (\mathit{Circle}, 2))] = \mathit{East}\\
\mathit{direction}\ [&(\mathit{Rectangular}, \mathit{Short}, 2, \mathit{Open}, \mathit{None}, (\mathit{Circle}, 1)),\\
&(\mathit{Hexagonal}, \mathit{Short}, 2, \mathit{Closed}, \mathit{Flat}, (\mathit{Triangle}, 1)),\\
&(\mathit{Rectangular}, \mathit{Long}, 3, \mathit{Closed}, \mathit{Flat}, (\mathit{InvertedTriangle}, 1))] = \mathit{East}\\
\mathit{direction}\ [&(\mathit{BucketShaped}, \mathit{Short}, 2, \mathit{Open}, \mathit{None}, (\mathit{Triangle}, 1)),\\
&(\mathit{DoubleRectangular}, \mathit{Short}, 2, \mathit{Open}, \mathit{None}, (\mathit{Triangle}, 1)),\\
&(\mathit{Ellipsoidal}, \mathit{Short}, 2, \mathit{Closed}, \mathit{Curved}, (\mathit{Diamond}, 1)),\\
&(\mathit{Rectangular}, \mathit{Short}, 2, \mathit{Open}, \mathit{None}, (\mathit{Rectangle}, 1))] = \mathit{East}\\
\mathit{direction}\ [&(\mathit{DoubleRectangular}, \mathit{Short}, 2, \mathit{Open}, \mathit{None}, (\mathit{Triangle}, 1)),\\
&(\mathit{Rectangular}, \mathit{Long}, 3, \mathit{Closed}, \mathit{Flat}, (\mathit{LongRectangle}, 1)),\\
&(\mathit{Rectangular}, \mathit{Short}, 2, \mathit{Closed}, \mathit{Flat}, (\mathit{Circle}, 1))] = \mathit{East}\\
\mathit{direction}\ [&(\mathit{Rectangular}, \mathit{Long}, 2, \mathit{Closed}, \mathit{Flat}, (\mathit{Circle}, 3)),\\
&(\mathit{Rectangular}, \mathit{Short}, 2, \mathit{Open}, \mathit{None}, (\mathit{Triangle}, 1))] = \mathit{West}\\
\mathit{direction}\ [&(\mathit{DoubleRectangular}, \mathit{Short}, 2, \mathit{Open}, \mathit{None}, (\mathit{Circle}, 1)),\\
&(\mathit{UShaped}, \mathit{Short}, 2, \mathit{Open}, \mathit{None}, (\mathit{Triangle}, 1)),\\
&(\mathit{Rectangular}, \mathit{Long}, 2, \mathit{Closed}, \mathit{Jagged}, (\mathit{Null}, 0))] = \mathit{West}\\
\mathit{direction}\ [&(\mathit{Rectangular}, \mathit{Long}, 3, \mathit{Closed}, \mathit{Flat}, (\mathit{LongRectangle}, 1)),\\
&(\mathit{UShaped}, \mathit{Short}, 2, \mathit{Open}, \mathit{None}, (\mathit{Circle}, 1))] = \mathit{West}\\
\mathit{direction}\ [&(\mathit{BucketShaped}, \mathit{Short}, 2, \mathit{Open}, \mathit{None}, (\mathit{Circle}, 1)),\\
&(\mathit{Rectangular}, \mathit{Long}, 3, \mathit{Closed}, \mathit{Jagged}, (\mathit{LongRectangle}, 1)),\\
&(\mathit{Rectangular}, \mathit{Short}, 2, \mathit{Open}, \mathit{None}, (\mathit{Rectangle}, 1)),\\
&(\mathit{BucketShaped}, \mathit{Short}, 2, \mathit{Open}, \mathit{None}, (\mathit{Circle}, 1))] = \mathit{West}\\
\mathit{direction}\ [&(\mathit{UShaped}, \mathit{Short}, 2, \mathit{Open}, \mathit{None}, (\mathit{Rectangle}, 1)),\\
&(\mathit{Rectangular}, \mathit{Long}, 2, \mathit{Open}, \mathit{None}, (\mathit{Rectangle}, 2))] = \mathit{West}.
\end{aligned}
$$

Experiment Setting We used one hidden node in a single hidden layer in this experiment. For the labelling function, the 1 of C encoding is used for the constants. Figure 4.4 shows the basic network components. Note that we have flattened the *load* component. This reduces the complexity of the individual networks without affecting the actual

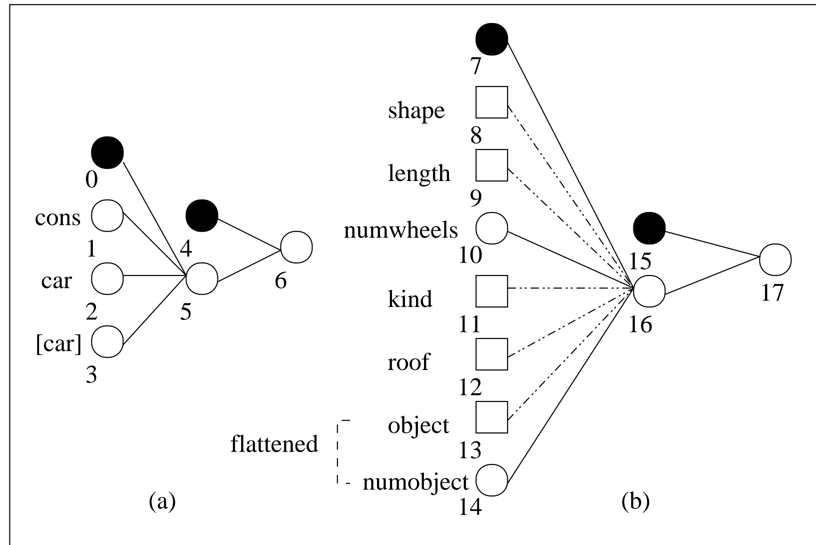


Figure 4.4: The trains network components. One node (excluding the bias node) was used in the hidden layer. The black nodes are bias nodes with input fixed at 1. All the nodes are numbered for referencing purposes. (a) The [*Car*] component. (b) The *Car* component.

information content. The square boxes represent a collection of nodes corresponding to the encoding scheme. The square box for Roof, for example, has 5 nodes. The dotted lines represent solid connections out of each node in a particular square box. Learning rate is set at 0.05, and momentum at 0.3. The weights are initialised at values very close to zero.

Experimental Result After about 250,000 epochs, the learner obtains 100% accuracy on the training set. The final weight solution is shown in Table 4.3.

Discussion Readers may note that the network component for sets and lists are quite similar, in fact, if the labelling function produces the same output for both *true* and *cons*, then they are identical. The architecture itself makes no distinction between the two. The difference lies in the way these components are used to construct the individual networks. For sets, the elements are always re-ordered (according to some arbitrary order) because the order does not matter. This preprocessing step saves the network from having to learn the equivalence relations corresponding to identical but differently ordered sets. Conversely, for lists, the elements are always presented in their original order because the order does matter. It is the order in which the elements of sets and lists are presented to the network that differentiates them from one another.

Sets can be thought of as a "degenerative" case of abstraction. In general, abstraction networks for other forms of finite lookup table, *e.g.*, multisets, are quite different from list networks because the return values for the elements of the lookup table are not all identical.

Table 4.3: The weight solution for the East-West problem.

i \ j	0	1	2	3	4	5
5	-3.36	-0.20	6.71	0.056	-	-
6	-	-	-	-	4.20	-8.61

i \ j	7	8	9	10	11	12	13	14	15	16
16	0.54	-2.09	0.37	0.14	1.13	1.14	1.24	-0.01	-	-
		0.90	-0.06		-0.68	0.01	-0.04			
		0.76				0.09	-0.73			
		0.87				-0.04	1.55			
		-0.06				-0.61	0.64			
		-0.03					-2.40			
							-0.07			
							-0.00			
							0.05			
17	-	-	-	-	-	-	-	-	-2.81	5.97

4.1.4 Interesting Trees (Binary Tree)

The problem is to determine whether a given binary tree is "interesting". The labelling of the trees is done at random.

Problem Specification We define the following type

$$BIntTree = BTree\ Int.$$

The task is to learn a function $intTree : BIntTree \to \Omega$. Here are the examples.

$$intTree\ (BNode\ (BNode\ Null\ 1\ Null)\ 2\ (BNode\ Null\ 3\ Null)) = \top$$
$$intTree\ (BNode\ (BNode\ (BNode\ Null\ 1\ Null)\ 2\ (BNode\ Null\ 3\ Null))\ 4\ Null) = \top$$
$$intTree\ (BNode\ (BNode\ Null\ 3\ (BNode\ Null\ 4\ Null))\ 5\ (BNode\ Null\ 10\ Null)) = \top$$
$$intTree\ (BNode\ (BNode\ (BNode\ Null\ 4\ Null)\ 5\ (BNode\ Null\ 7\ Null))\ 10\ (BNode\ (BNode\ Null\ 11\ Null)\ 12\ Null)) = \top$$
$$intTree\ (BNode\ (BNode\ (BNode\ (BNode\ Null\ 2\ Null)\ 3\ Null)\ 4\ Null)\ 5\ Null) = \top$$
$$intTree\ (BNode\ (BNode\ Null\ 7\ Null)\ 5\ (BNode\ Null\ 8\ Null)) = \bot$$
$$intTree\ (BNode\ (BNode\ (BNode\ Null\ 8\ Null)\ 7\ (BNode\ Null\ 9\ Null))\ 6\ (BNode\ Null\ 5\ (BNode\ Null\ 4\ Null))) = \bot$$
$$intTree\ (BNode\ Null\ 5\ (BNode\ Null\ 6\ (BNode\ Null\ 4\ (BNode\ Null\ 3\ Null)))) = \bot$$
$$intTree\ (BNode\ (BNode\ Null\ 3\ Null)\ 2\ (BNode\ Null\ 1\ Null)) = \bot$$
$$intTree\ (BNode\ (BNode\ (BNode\ Null\ 9\ Null)\ 10\ Null)\ 2\ (BNode\ (BNode\ Null\ 8\ (BNode\ Null\ 9\ Null))\ 10\ Null)) = \bot$$

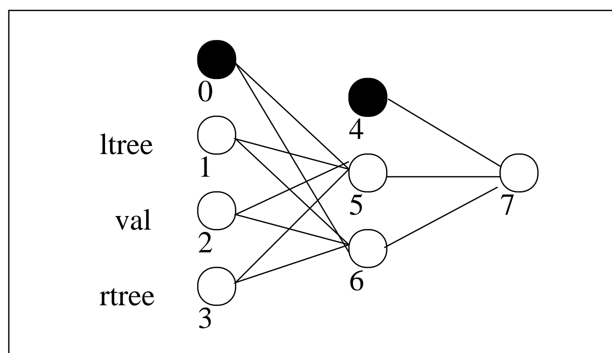


Figure 4.5: The binary tree network component. Two nodes (excluding the bias node) were used in the hidden layer. The black nodes are bias nodes with input fixed at 1. All the nodes are numbered for referencing purposes.

Table 4.4: The weight solution for the binary tree component.

i \ j	0	1	2	3	4	5	6
5	4.82518	0.91146	-2.11914	17.7595	-	-	-
6	-16.4395	-7.64127	0.302936	23.6885	-	-	-
7	-	-	-	-	6.5462	-16.8636	11.3863

Experiment Setting We used two hidden nodes in a single hidden layer for our only network component, which is shown in Figure 4.5. Direct encoding is used for the labelling function. Learning rate is set at 0.05, and momentum at 0.05. The weights are initialised randomly at values close to zero.

Experimental Result After slightly less than 100,000 epochs, the learner obtains 100% accuracy on the dataset. Table 4.4 shows the weight solution obtained.

4.1.5 Bongard 47 (Directed Graph)

This is problem 47 in the book [Bon70, p.229] by Bongard on pattern recognition. This problem is illustrated in Figure 4.6.

Problem Specification As usual, we must first decide on a suitable type to represent a Bongard diagram. For this, we have chosen to use a directed graph. The geometric figures in the diagram, which can be circles, triangles, or squares, are the nodes of the graph and there is a directed edge from shape s_1 to shape s_2 if and only if s_2 is located inside s_1.

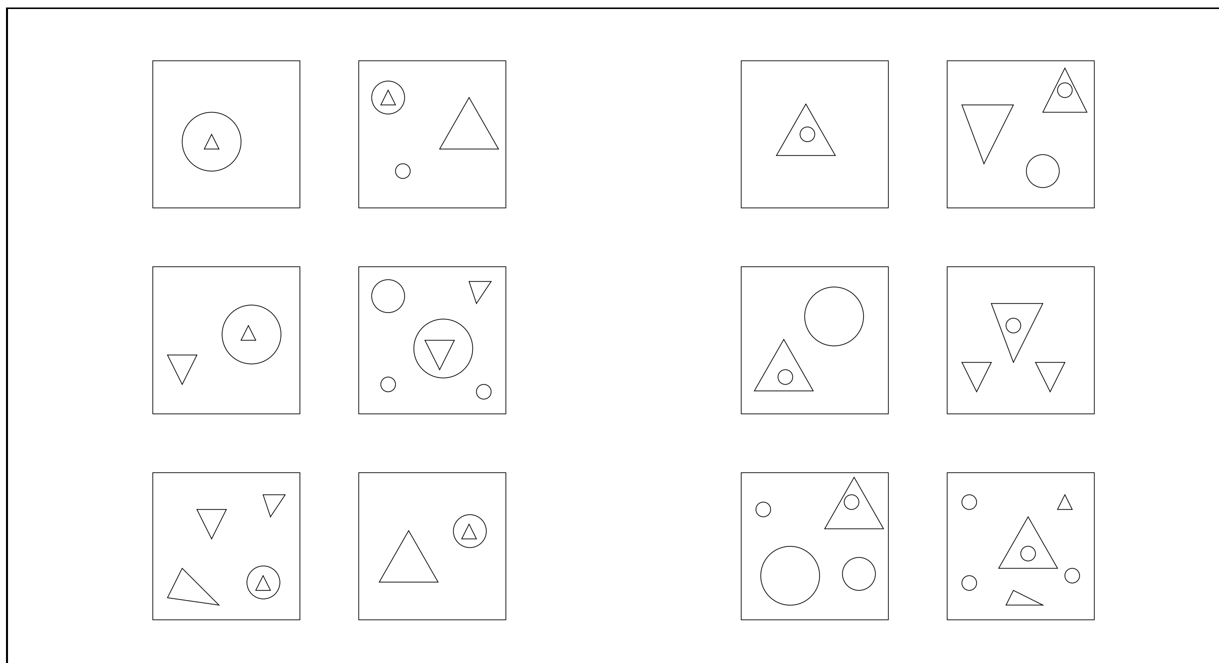


Figure 4.6: Bongard: Examples in Class1 are on the left; those in Class2 are on the right.

We introduce the types *Shape* and *Class* as follows.

$$Circle, Triangle, Square : Shape$$
$$Class1, Class2 : Class.$$

For convenience, we also introduce the following type synonyms.

$$Label = Nat$$
$$Digraph\ v\ e = \{Label \times v\} \times \{(Label \times Label) \times e\}$$

In this problem, there is no information on the edges of the graph, so we use the type *1* as a default for the type of this information and the empty tuple () is used for all these values. The type *Diagram* is defined as follows.

$$Diagram = Digraph\ Shape\ 1.$$

Vertices contain information of type *Shape*, while edges contain information of type *1* (which is equivalent to having no information at all). Thus the term

$$(\{(1, Circle), (2, Triangle)\}, \{((1, 2), ())\})$$

represents the diagram with a circle and a triangle, where the triangle is inside the circle.

We want to learn the definition of the function *class* with signature:

$$class : Diagram \rightarrow Class.$$

Here are the training examples.

$$
\begin{aligned}
&class\ (\{(1, Circle), (2, Triangle)\}, \{((1, 2), ())\}) \ = \ Class1 \\
&class\ (\{(1, Circle), (2, Triangle), (3, Triangle), (4, Circle)\}, \{((1, 2), ())\}) \ = \ Class1 \\
&class\ (\{(1, Triangle), (2, Circle), (3, Triangle)\}, \{((2, 3), ())\}) \ = \ Class1 \\
&class\ (\{(1, Circle), (2, Triangle), (3, Circle), (4, Triangle), (5, Circle), (6, Circle)\}, \\
&\qquad \{((3, 4), ())\}) \ = \ Class1 \\
&class\ (\{(1, Triangle), (2, Triangle), (3, Triangle), (4, Circle), (5, Triangle)\}, \\
&\qquad \{((4, 5), ())\}) \ = \ Class1 \\
&class\ (\{(1, Triangle), (2, Circle), (3, Triangle)\}, \{((2, 3), ())\}) \ = \ Class1 \\
&class\ (\{(1, Triangle), (2, Circle)\}, \{((1, 2), ())\}) \ = \ Class2 \\
&class\ (\{(1, Triangle), (2, Triangle), (3, Circle), (4, Circle)\}, \{((2, 3), ())\}) \ = \ Class2 \\
&class\ (\{(1, Circle), (2, Triangle), (3, Circle)\}, \{((2, 3), ())\}) \ = \ Class2 \\
&class\ (\{(1, Triangle), (2, Circle), (3, Triangle), (4, Triangle)\}, \{((1, 2), ())\}) \ = \ Class2 \\
&class\ (\{(1, Circle), (2, Triangle), (3, Circle), (4, Circle), (5, Circle)\}, \{((2, 3), ())\}) \ = \ Class2 \\
&class\ (\{(1, Circle), (2, Triangle), (3, Triangle), (4, Circle), (5, Circle), (6, Triangle), (7, Circle)\}, \\
&\qquad \{((3, 4), ())\}) \ = \ Class2
\end{aligned}
$$

Experiment Setting We used four nodes in a single hidden layer for all five components. Figure 4.7 shows the five components used for this experiment. Note that the two vertex labels of an edge have been flattened. 1 of C encoding is used for the shape values, and direct encoding is used for everything else. Learning rate is set at 0.01, with zero momentum. The weights are initialised randomly at values close to zero.

Experimental Result After about 100,000 epochs, the learner obtains 100% accuracy on the dataset, Table 4.5 shows the final weight solution for each component.

Discussion This result is perhaps rather surprising. The mathematical definition of a directed graph, a 2-tuple where the first element is the set of labelled vertices in the graph, and the second element is the set of directed edges, does not give a strong clue to the underlying structural properties of a directed graph. Structural information can be extracted from the set of edges, but that requires the network to realise that the labels in the ordered pairs are indeed pointers to vertices. It is inconceivable how that function can be captured.

The network nonetheless achieves 100% accuracy on the training set. There are two ways to interpret this result: 1) By some miraculous construct, the network somehow captures the underlying target function. 2) The network merely performs a rote learning in this case, remembering all the diagrams.

For interest, a solution for this problem given in [BGCL01] is "A diagram is in class 2 if it contains a circle inside something; otherwise, it is in class 1."

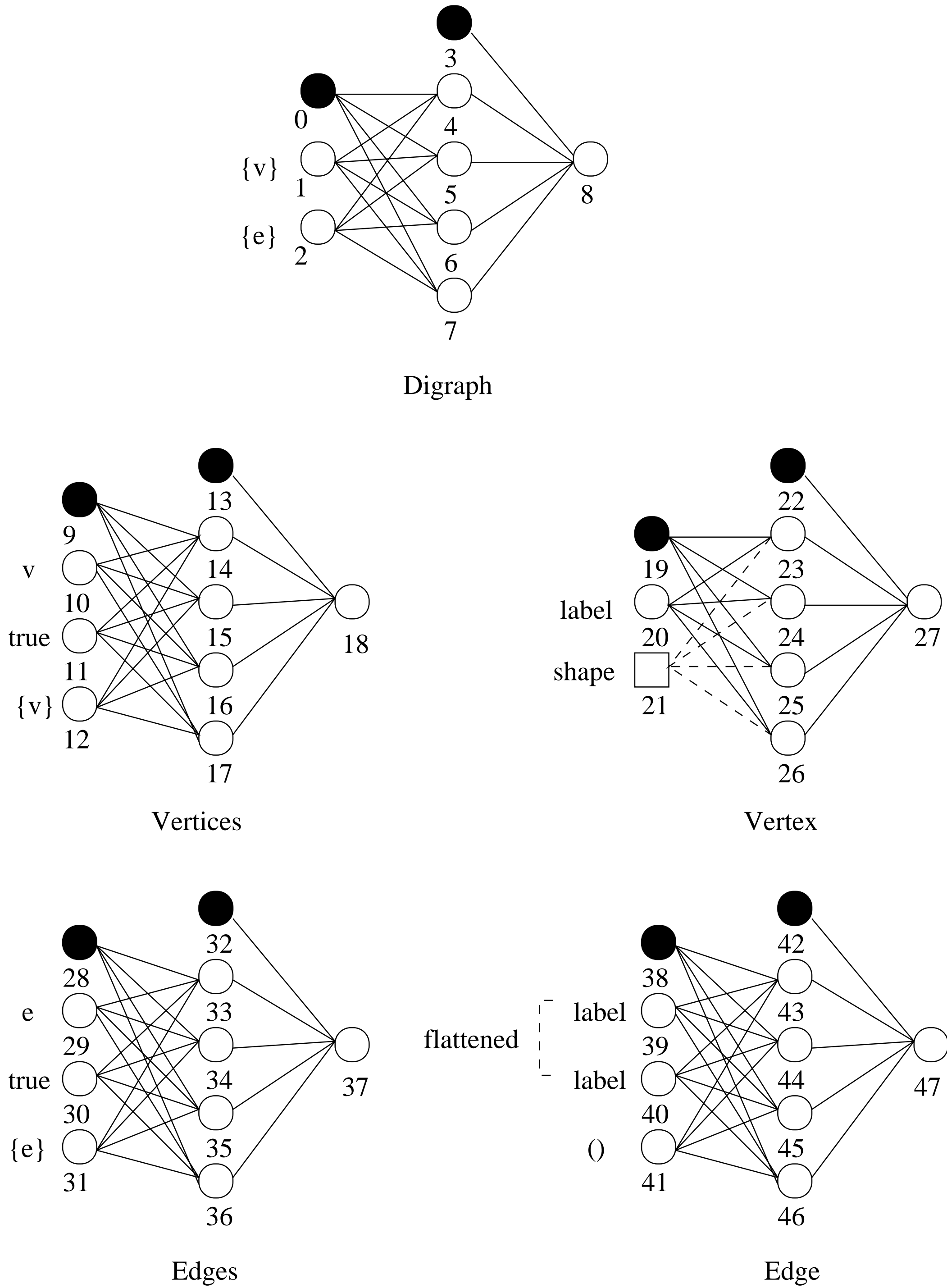


Figure 4.7: Network components for the Bongard problem. Four nodes (excluding the bias node) were used in the hidden layer. The black nodes are bias nodes with input fixed at 1. All the nodes are numbered for referencing purposes.

Table 4.5: The weight solution for the Bongard experiment.

i \ j	0	1	2	3	4	5	6	7	
4	0.54	-1.39	-0.94	-	-	-	-	-	
5	0.78	-3.82	0.80	-	-	-	-	-	
6	0.59	1.68	-0.28	-	-	-	-	-	
7	1.48	-5.24	0.89	-	-	-	-	-	
8	-	-	-	-2.54	0.87	4.11	-2.96	5.91	

i \ j	9	10	11	12	13	14	15	16	17
14	1.04	-2.76	-0.44	2.90	-	-	-	-	-
15	-0.26	-2.10	0.38	2.40	-	-	-	-	-
16	0.41	-1.82	-0.49	2.32	-	-	-	-	-
17	0.47	-3.40	0.67	2.86	-	-	-	-	-
18	-	-	-	-	4.23	-3.51	-2.67	-2.13	-3.97

i \ j	19	20	21	22	23	24	25	26	
23	0.33	-0.97	-2.19 1.20	-	-	-	-	-	
24	-0.69	-0.76	-0.51 0.60	-	-	-	-	-	
25	0.89	-2.36	-1.29 2.28	-	-	-	-	-	
26	-0.28	-1.06	-0.26 -0.33	-	-	-	-	-	
27	-	-	-	2.51	-2.29	-0.87	-3.24	-0.62	

i \ j	28	29	30	31	32	33	34	35	36
33	0.80	-0.29	-0.68	0.16	-	-	-	-	-
34	-0.81	-0.49	-0.17	0.13	-	-	-	-	-
35	0.90	-0.37	0.01	1.00	-	-	-	-	-
36	-0.37	-0.96	-0.17	-0.51	-	-	-	-	-
37	-	-	-	-	0.93	-0.38	-0.35	0.00	-0.36

i \ j	38	39	40	41	42	43	44	45	46
43	0.84	-0.29	-0.65	0.16	-	-	-	-	-
44	-0.79	-0.48	-0.15	0.13	-	-	-	-	-
45	0.91	-0.36	0.02	1.00	-	-	-	-	-
46	-0.34	-0.96	-0.14	-0.51	-	-	-	-	-
47	-	-	-	-	0.56	-0.56	-0.44	-0.24	-0.47

4.2 Three Less Easy Problems

The section presents the experimental results of applying term reduction networks to three larger problems. The first is an artificial dataset; the second is a benchmark problem in machine learning; and the third is an open problem in the area of predictive toxicology.

4.2.1 Binary Search Trees

Here, I introduce a new problem which illustrates a few points rather nicely. The problem is to determine whether a given binary tree is a search tree. This problem is interesting because there is a recursive function which the single network component needs to learn: A binary tree rooted at x is a search tree if and only if both the left and right subtrees are search trees, and the largest element in the left subtree is smaller or equal to x, and the smallest element in the right subtree is strictly larger than x.

The dataset consists of 500 randomly generated binary trees, with 250 positive instances and 250 negative instances. Every individual in the dataset can have anything from 3 to 42 nodes, and the integers at the nodes can take on any value between 1 and 20 inclusive (there is nothing magical about the four numbers, they are arbitrary choices). The dataset is noise-free. Several sample binary trees in the dataset can be found in Appendix B. The complete dataset can be found in `http://discus.anu.edu.au/~kee/bst`.

Problem Specification We define the following type

$$BIntTree = BTree\ Int$$

for binary trees with integer-bearing nodes. The task is to learn a function with the signature $searchTree : BIntTree \to \Omega$.

Experiment Setting We used a single hidden layer with 12 nodes for this experiment. Figure 4.8 shows the network component. Direct encoding is used for the integers. Learning rate is set at 0.01 and momentum at 0.01. The weights are initialised with random values using $a = 2$. The a here refers to the parameter a in Bottou's formula given in page 27.

The performance of the learner was measured using a ten-fold cross-validation experiment, which was done as follows. The examples are partitioned randomly into ten different groups, each with a similar class distribution as the original dataset. Learning is conducted on nine groups, and the resulting solution function is applied to the last unseen group. This process is iterated ten times on the ten different groups. The generalisation accuracy is estimated using the average of the ten measured accuracy on the test groups. For more details on the statistical significance of this technique, see [Koh95].

Experimental Result On a ten-fold cross-validation, the learner obtains an average of 87.2% accuracy on the validation sets. The is only slightly lower than the 87.9% average accuracy on the training sets, reflecting good generalisation performance.

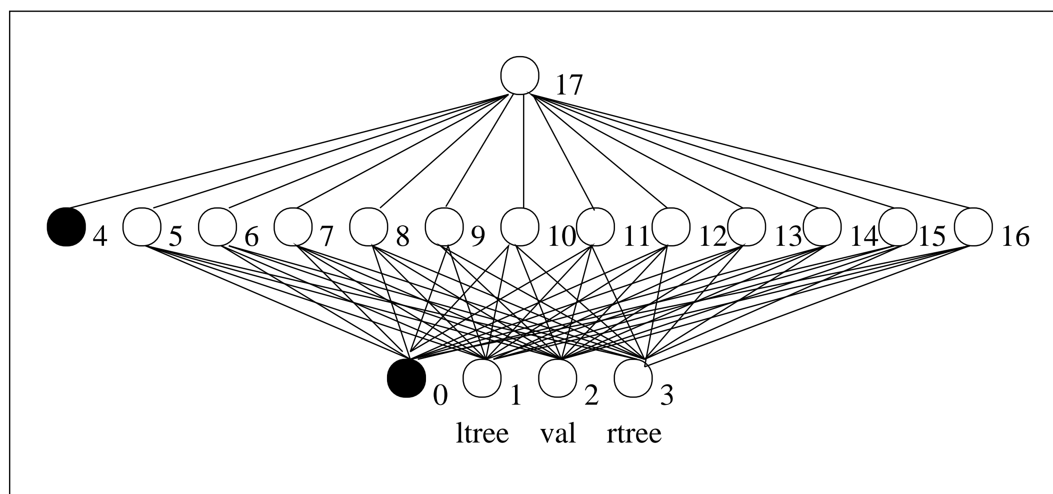


Figure 4.8: The binary search tree network component.

Discussion In principle, a symbolic learner with appropriate background theory can achieve 100% accuracy on the dataset. However, this does not make the result obtained by term reduction networks any less remarkable. A symbolic learner depends on having all the basic concepts made available in the background theory. Term reduction networks do not have such luxury and everything needs to be constructed from scratch.

4.2.2 Musk

The Musk problem, proposed in [DLLP97] is a multiple-instance type problem. This class of problems is of interest to us here because it cannot be easily solved by conventional attribute-value language learners. Briefly, in the multiple-instance learning setting, each individual is a set of some elements. An individual belongs to a certain class if and only if it contains certain type of elements. The function to be learned maps a set to a random variable, the class of the individuals.

Neural networks have been used to solve this class of problems before. [RR00] suggested an extension of neural networks to multiple-instance type problems and reported excellent results in the Musk problem. However, the proposed solution is not entirely satisfactory because the resulting network explicitly caters for multiple-instance type problems. It is not at all clear that it can be generalised. Here, we use our generic network architecture to solve this challenging application.

Problem Specification In essence, the problem is to determine whether a molecule has a musk odour. Each molecule contains many different conformations and, presumably, only one conformation is responsible for the activity. Each conformation is a tuple of 166

floating point numbers, the first 162 of which represent the distance in angstroms from some origin in the conformation out along a radial line to the surface of the conformation, and the remaining 4 numbers represent the position of a specific oxygen atom.

For our experiments, we used a discretised version of the original musk datasets. This discretisation was done as follows: For each of the 166 attributes, we calculated the mean m and the standard deviation sd of the values occurring in the data. We then built up intervals centered on the mean, taking the width of each interval to be one standard deviation, and assigned integral labels to the intervals, so that interval 0 centred on the mean is $[m-sd/2, m+sd/2]$, interval 1 is $[m+sd/2, m+3sd/2]$, interval -1 is $[m-3sd/2, m-sd/2]$ and so on. We chose to use 13 intervals in total, labeled -6 through 6, as this covers most of the distribution and gives adequate resolution. The outermost intervals, -6 and 6, were extended below and above respectively to cover any outlying points [BGCL01].

From the above, we obtain the following type information:

$$-6, -5, \ldots, 5, 6 : \mathit{Distance}$$
$$\mathit{Conformation} = \mathit{Distance} \times \cdots \times \mathit{Distance}$$
$$\mathit{Molecule} = \{\mathit{Conformation}\}.$$

Here, *Conformation* is a product of 166 *Distance* components. The task is to learn a function with the signature $\mathit{musk} : \mathit{Molecule} \to \Omega$.

Experiment Setting For all the experiments, the number of hidden nodes for the $\{Conformation\}$ component are fixed at 2. The network architectures for all the experiments differ only in the number of hidden nodes for the conformation component. All network components are fully connected. Figure 4.9 shows the general form of network components used to built up the individual networks.

Various ways to achieve good generalisation were tried. Among them early stopping with validation set [Sar95] and early stopping. In the former case, a subset of the training data is chosen to be the validation set. Training is conducted on the remaining training examples. Learning stops when the predictive accuracy on the validation set starts to drop. In the latter case, a target accuracy is chosen prior to training. Learning stops after that target has been achieved on the training data. The generalisation accuracy reported is estimated using ten-fold cross-validation.

Experimental Result There are two datasets: musk-1 with 92 molecules and musk-2 with 102 molecules. Musk-2 is computationally much more taxing than musk-1 because each molecule in musk-2 contains far more conformations compared to molecules in musk-1.

Table 4.6 shows the summaries of results obtained for musk-1. The second set of results with high number of hidden nodes were obtained with early stopping using validation sets. Table 4.7 shows the summary of results obtained for musk-2. These experiments were conducted with early stopping without a validation set. The learner stops when the accuracy on the training set goes consistently above 90% for a fixed number of epochs.

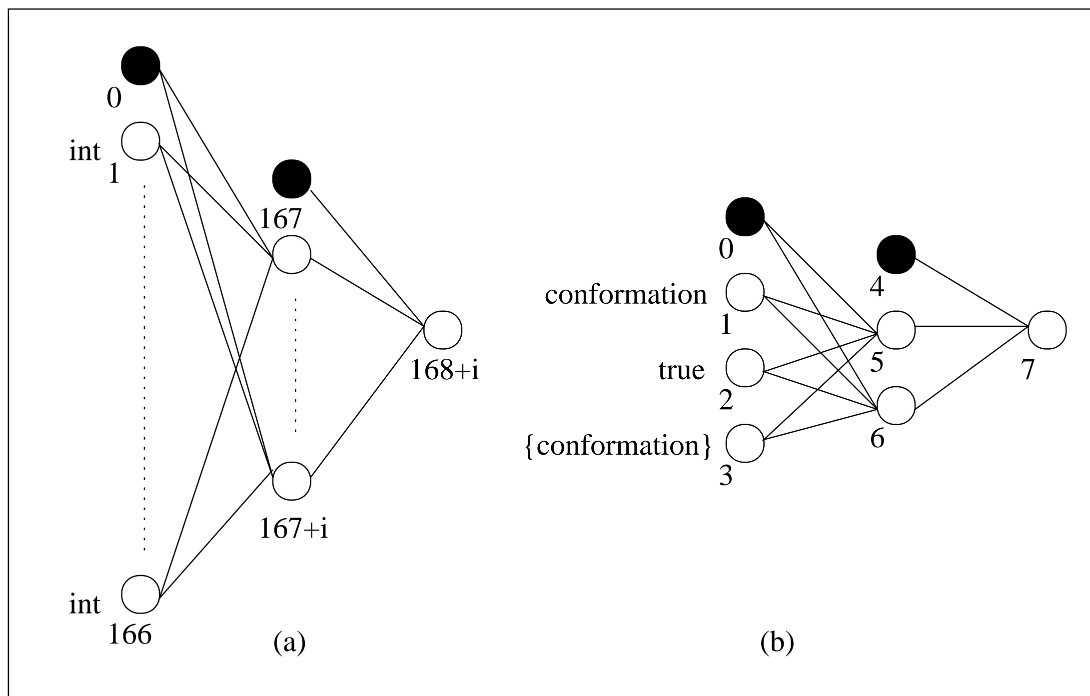


Figure 4.9: The musk network components.

Table 4.6: Summary of musk-1 results : Accuracy vs Hidden nodes.

Hidden nodes	1	2	3	4	5	6	7	8	9
Accuracy (%)	73.53	73.55	71.50	70.89	72.55	71.89	70.38	70.83	70.33
Hidden nodes	10	20	30	40	50	60	70	80	90
Accuracy (%)	77.22	77.22	79.55	76.11	76.11	76.11	76.22	76.00	76.22

Table 4.7: Summary of musk-2 results : Accuracy vs Hidden nodes. (early stopping without validation set)

Hidden nodes	2	3	4	5	6
Accuracy (%)	77.45	82.50	85.05	85.30	82.36

Discussion Here, we compare our results with that obtained by other researchers in the field. Table 4.8 (adapted from [RR00]) shows the summary of major results obtained by different groups on the musk dataset.

The tangent distance and dynamic reposing techniques require computation of the

Table 4.8: Known results for the musk problem.

Method	Musk1 (%)	Musk2 (%)
Iterated-discrim APR	92.4	89.2
GFS Elim-kde APR	91.3	80.4
GFS Elim-count APR	90.2	75.5
GFS All-positive APR	83.7	66.7
All-positive APR	80.4	72.6
Simple-back-propagation - 1 instance / molecule	75.0	67.7
C4.5 (pruned)	68.5	58.8
1-nearest Neighbour (Euclidean Distance)		75
Neural Network (Standard Poses)		75
1-nearest Neighbour (Tangent Distance)		79
Neural Network (Dynamic Reposing)		91
Higher-order Decision Tree		83.5
Multi Instance Neural Network	88	82
Term Reduction Network ↩	79.55	85.30

molecular surface. This information is not available in the feature vectors in the dataset we used, and hence a direct comparison cannot be made.

For a general-purpose learning system, term reduction network performed admirably well compared to others on the musk-2 dataset, although the same thing cannot be said of musk-1. This is inconsistent with all other methods, all of which show better results on musk-1 than on musk-2. One possible explanation is in the discretised process performed on the dataset. By doing the discretisation, we lose some information content but gain more regularity in the dataset. In most circumstances, the net effect is that the regularity so-obtained more than compensates for the loss of original information. This may not be the case in musk-1, where there are not many conformations in each individual. It is possible that the loss of information cannot be offset by the gain in the regularity of input in the case of musk-1.

On the surface, the results look promising, but there is one problem. There is no easy way to tell whether the solutions obtained are indeed multiple-instance solutions. In all likelihood, they are not! If we allow ourselves to view neural networks as black boxes, the probably approximately correct framework developed in Valiant's seminal paper [Val84] assures us that our networks can be trusted with high confidence to respond correctly to new unseen individuals drawn from the same distribution since the training sets are large compared to the degrees of freedom in the networks [BH89]. In that sense, the actual function implemented by the network does not matter.

Having said that, we can of course be pedantic and demand that a solution satisfying the multiple-instance constraint be produced. One way to achieve this is to change the topology of the $\{Conformation\}$ component to the one shown in Figure 4.10, and fix the weights as follows: $w_{40} = -x/2, w_{41} = x, w_{43} = x$ for some positive integer x. By changing

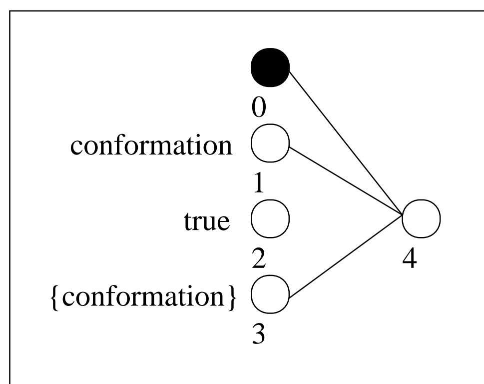


Figure 4.10: The {*Conformation*} network component.

the gradient of the squashing function to approximate a step function, we can force the output at each unit to take on values very close to either 0 or 1, and the component shown in 4.10 essentially becomes a logical OR gate. The effect of using such a component to construct the individuals is to perform a logical OR operation on all the elements in the set. The network returns true if and only if there exists at least one element in the set with a certain property, *i.e.,* the final solution must be a multiple-instance solution. We must choose a value for x initially, and the weights for all instances of the {*Conformation*} component stay fixed through-out training. The weights for the *Conformation* network instances are of course allowed to change. Such pre-initialised networks are in general harder to train, because the errors can not be efficiently propagated to the later layers in the *Conformation* component instances due to the fixed weights in the earlier layers. Preliminary experiments suggest that accuracies in the region of 76% on the musk-2 dataset can be obtained.

There are of course other (better) ways of introducing the constraint. However, in general, it is quite hard to simulate, with good precision, symbolic concepts using subsymbolic learners like neural networks.

4.2.3 The 2000-1 Predictive Toxicology Challenge

This section discusses the result of applying term reduction networks to an important real-world quantitative structure-activity (QSAR) problem - The 2001 Predictive Toxicology Challenge (PTC). The challenge was to obtain models that can predict the outcome of biological tests for the carcinogenicity of chemical compounds using information related to chemical structures only [HKKe01].

There are a total of 417 molecules in the dataset, each labelled with a classification for four different chemical bioassays conducted on male rats, female rats, male mice and female mice. The dataset was built up over many years from experiments conducted at the US National Toxicology Program (`http://ntp-server.niehs.nih.gov`). As a result,

there is some inconsistencies in the classification labellings used. For each of the four types of experiments, a molecule can have any one of the following labels: P (Positive), N (Negative), E (Equivocal), CE (Clear Evidence), SE (Some Evidence), EE (Equivocal Evidence), NE (No Evidence) and IS (Inadequate Study). Figure 4.2.3 (courtesy of A.W. Slater) shows four sample molecules from the dataset. In addition to the explicit structures of the molecules, seven sets of feature vectors containing chemical descriptors of various kinds are available from the initial data engineering phase. The complete dataset and related information can be found at the PTC home page [PTC01].

All the submissions made to the challenge (including one from the author, using a different approach) can be found in a compiled volume in [HKKe01]. It is generally agreed that the PTC is an extremely hard problem, and the results obtained so far do not look at all promising. The submissions are evaluated using ROC analysis, an effective method to evaluate the relative performance of learning systems by separating the issues of classifier performance from specific class and cost distributions (see [PF01] for an introduction). The results of the analyses can be found at the PTC home page. For simplicity, we shall continue to measure our performance in terms of accuracy, which assumes equal cost for both false positive and false negative predictions.

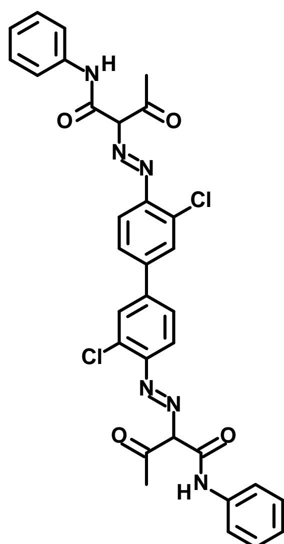
We used the Female Rat dataset for our experiments here, as it is the only one that looks remotely learnable from the ROC analyses. The labelling inconsistencies are handled as follows: All molecules with E, EE and IS as labels are discarded. Molecules with P, CE and SE as labels are collected in a group and labelled Positive. Molecules with N and NE as labels are collected in a second group and labelled Negative. The resulting dataset contains 351 molecules, with 121 positive examples and 230 negative examples. We chose not to use the feature vectors here.

Problem Specification The molecules can best be represented using undirected graphs. The question is, how do one represent graphs using basic terms? There are several ways to do this. We can use a standard mathematical definition of an undirected graph $G = (V, E)$, where V is the set of all the vertices of the graph, each uniquely labelled using a natural number; and E the set of all the edges of the graph, each represented as an unordered pair of the two vertices it connects. Alternatively, one can also represent graphs using adjacency lists and adjacency matrices.

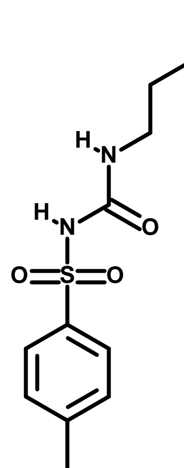
Here, we use a slightly different version of the first representation. We introduce the following constants for the elements and the class of edges.

$$As, Au, B, Ba, \ldots, S, Sn, Te, Zn : \mathit{Element}$$
$$-, =, \#, + : \mathit{Bond}$$

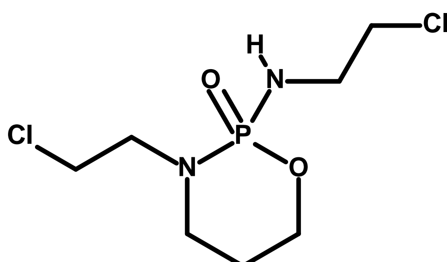
The four constants for *Bond* represents respectively single bond, double bond, triple bond and resonant bond.



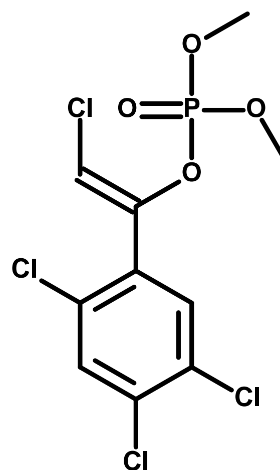
Tag Name: TR030
MR=N, FR=N,
MM=N, FM=N



Tag Name: TR031
MR=N, FR=N,
MM=N, FM=N



Tag Name: TR032
MR=N, FR=P,
MM=N, FM=P



Tag Name: TR033
MR=N, FR=P,
MM=P, FM=P

Figure 4.11: Sample molecules in The 2000-1 Predictive Toxicology Challenge.

We also introduce the following type synonyms.

$$\begin{aligned} \mathit{Label} &= \mathit{Nat} \\ \mathit{Vertex} &= \mathit{Label} \times \mathit{Element} \\ \mathit{Molecule} &= \{\mathit{Vertex}\} \times \{\mathit{Vertex} \times \mathit{Vertex} \times \mathit{Bond}\} \end{aligned}$$

Note that two changes have been made to the standard definition of a graph. Instead of using labels as pointers to the vertices, we have duplicated vertex information everywhere they occur. This is done to boost information content. As pointed out earlier in Section 4.1.5, it is difficult to see how the reduction networks can work out the connection between the labels and the information at the vertices without having direct weighted links between them. Earlier experiments with the standard representation confirmed this.

The second change is that in place of unordered pairs, which can be represented using a multiset with cardinality 2, ordered pairs are used for the edges. This is done to simplify the architecture of the individual networks, with the aim that faster convergence can be obtained. To counter the possible ill-effects of the artificially introduced ordering, the edges are preprocessed so that the label of the first vertex is always smaller than the label of the second vertex. The following is an example from the dataset, molecule TR032 as shown in Figure 4.2.3.

$$\begin{aligned} \mathit{TR032} = (\{&(1, c), (2, c), (3, cl), (4, n), (5, c), (6, c), (7, c), (8, o), (9, p), \\ &(10, o), (11, n), (12, c), (13, c), (14, cl), (15, h), (16, h), (17, h), \\ &(18, h), (19, h), (20, h), (21, h), (22, h), (23, h), (24, h), (25, h), \\ &(26, h), (27, h), (28, h), (29, h)\}, \\ \{&(11, n, 12, c, -), (12, c, 13, c, -), (13, c, 14, cl, -), (9, p, 11, n, -), \\ &(9, p, 10, o, =), (1, c, 15, h, -), (1, c, 16, h, -), (2, c, 17, h, -), \\ &(2, c, 18, h, -), (5, c, 19, h, -), (1, c, 2, c, -), (11, n, 25, h, -), \\ &(12, c, 26, h, -), (12, c, 27, h, -), (13, c, 28, h, -), (13, c, 29, h, -), \\ &(5, c, 20, h, -), (6, c, 21, h, -), (6, c, 22, h, -), (7, c, 23, h, -), \\ &(7, c, 24, h, -), (2, c, 3, cl, -), (1, c, 4, n, -), (4, n, 5, c, -), \\ &(5, c, 6, c, -), (6, c, 7, c, -), (7, c, 8, o, -), (4, n, 9, p, -), (8, o, 9, p, -)\}) \end{aligned}$$

The task is to learn a function with the signature $\mathit{carcinogenic} : \mathit{Molecule} \to \Omega$.

Experiment Setting A single hidden layer with eight nodes were chosen for the internal architectures of the component networks. Learning rate is set at 0.02 and momentum at 0.05. The weights are initialised with random values using $a = 5$. The 1 of C scheme is used to encode the elements, and direct-scaled encoding is used for the labels. Learning is configured to stop after an accuracy of 70% is achieved on the training set. The learner stops after 20,000 epochs if the target accuracy cannot be achieved. Again, a ten-fold cross-validation is conducted to estimate the generalisation performance of the learner.

Experimental Result The learner achieved an average generalisation accuracy of 61.85% on the dataset. The average accuracy on the training set is 69.96%. One of the ten folds failed to learn any meaningful concept, returning Negative, which is the majority class, for all molecules.

It is possible to set a higher target accuracy for the training phase, up to around 75%, but later experiments suggest that will not have any positive effect on the generalisation performance. In fact, the learner is likely to overfit the training data and return worse average accuracy on the test data.

Discussion The results obtained from the experiments here are consistent with results from previous attempts by other groups on the dataset (see [HKKe01]). In terms of accuracy, 65% seems to be the highest attainable, even when the thousands of features available are used, either independently or jointly with the molecular structures.

I put this failure down to two factors. Firstly, the fact that no one has yet achieve a respectable result on the dataset points to the difficulty of the problem and potential deficiencies in the original dataset. Expert advice seems to indicate this is indeed the case [Ric01]. The second factor, which I believe is the major reason, lies with the limitation of our graph representation. Our chosen representation fails to capture in an explicit fashion all the structural information of a graph, thus making the task of finding sub-structures responsible for carcinogenic activity, assuming such concepts exist, a very difficult one.

4.3 An Analysis of the Results

The results obtained using term reduction networks look very promising. The Five Easy Pieces convincingly illustrate the rich representational power of the learning system, showing its potential applicability to a wide range of problems. It has been shown that many data types including tuples, lists, sets, trees and combinations of these data types can be supported with ease. There is, however, one significant limitation in the representation language. The system cannot handle graphs very satisfactorily. This is because it is hard to find a term representation that can capture the structure of a graph in an explicit fashion. For data types like lists and trees, the structures of the terms, and hence the structures of the neural networks constructed from them, reflect the actual structures of the individuals they represent. Not so with graphs. The $G = (V, E)$ representation is quite clearly not the best of choices. It remains to be seen whether other ways to represent graphs, for example adjacency lists and adjacency matrix can help alleviate this problem. Having said that, it should be pointed out that this problem is more a theoretical criticism than a true limitation. Empirical studies have certainly shown that the modified $G = (V, E)$ representation with duplicated vertex data contains sufficient information for the learner to pick up complex concepts like the connectedness of a graph.

The Three Less Easy Problems clearly demonstrate that term reduction networks can be trained to give excellent generalisation performances. But good generalisation is not an inherent property of term reduction networks. In fact, term reduction networks, like

most neural networks, have a strong tendency to overfit the data because of their immense approximation capacities. This is evident from the results obtained for Interesting Trees and Bongard 47. In the former case, the trees are randomly assigned a label. In the latter case, the term representations do not reflect the true structures of the individuals. But in both cases, the learner managed to obtain 100% accuracy. Clearly, the learner simply performed a rote learning in the two cases, memorising all the training examples.

One other observation is that term reduction networks are extremely sensitive to the initial conditions. The initial topology and random weight values have huge impacts on the learnability of the networks. The author has limited experience with conventional neural networks, it is not clear whether the degree of sensitivity experienced here is consistent with that for conventional neural networks, or that the difficulties we had are peculiar to term reduction networks.

Chapter Notes

The type and data representations for Tennis, Keys, The East-West Challenge, Bongard 47 and Musk all come from [BGCL01].

It is possible to store the mind with a million facts
and still be entirely uneducated.
Alec Bourne

Chapter 5

Related Work

> Mathematical discoveries, small or great are never born of spontaneous generation.
> They always presuppose a soil seeded with preliminary knowledge
> and well prepared by labour, both conscious and subconscious.
> *Jules Henri Poincaré*

5.1 Introduction

In this chapter, I trace the development of term reduction networks to its direct ancestors and siblings. The integration of symbolic and subsymbolic learning models has been and still is an active and fertile research area in artificial intelligence. Some early works on this subject can be found in a compiled volume in [Hin90b]. In it, many innovative techniques were explored by different authors. Among them, two articles that introduce some novel and universal concepts for adaptive processing of structured data are of particular interest to us here. The first is [Hin90a], which introduces the concept of reduced descriptions. The second is [Pol90], which introduces Recursive Auto-associative Memory, one of the first neural network architectures to clearly demonstrate ability to learn from structured data. I briefly describe the two in the following two sections. This is followed by descriptions of three architectures that were built on it.

5.2 Hinton's Reduced Descriptions

Lying at the heart of most existing term-reduction-network-like architectures is the concept of *reduced representation*, a term first introduced by Geoffrey Hinton in [Hin90a] to denote a pointer-like "tag" that is useful in the construction of part-whole hierarchies. The best way to understand this idea is through an illustration. Consider the binary tree shown in Figure 5.1(a). It is a recursive compositional structure and hence has a part-whole hierarchy, in the sense that a node can be both a part of a tree and the whole of a tree. For example, the tree rooted at node B is both the whole of a tree and a part of a bigger tree

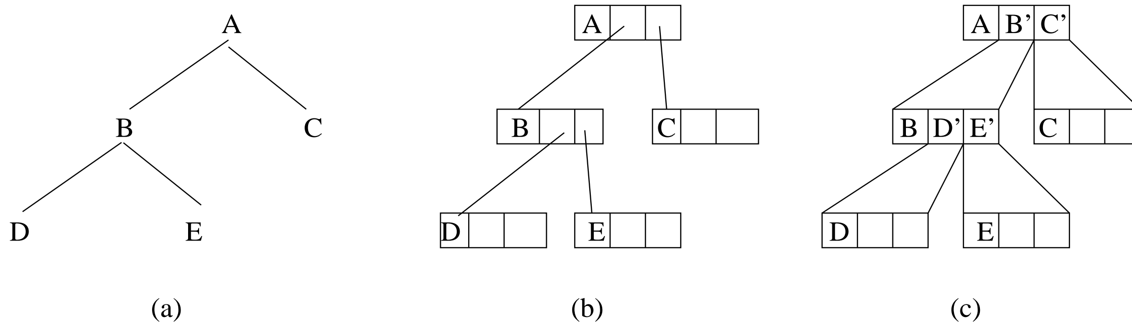


Figure 5.1: (a) A simple binary tree. (b) A typical binary tree data structure on von Neumann machines implemented using pointers. (c) A possible connectionist representation.

rooted at A. To figure out how such a structure can be represented in a neural network, we first examine how that can be done in conventional von Neumann machines. Figure 5.1(b) shows a typical pointer-based data structure that is commonly used to represent binary trees. In trying to build a similar connectionist structure, we immediately run into a problem, we do not have a direct analogue of pointers (or randomly addressable memory cells) in connectionist networks. Can we simulate a similar facility? The answer is fortunately yes.

Consider the neural network shown in Figure 5.1(c). Each square denotes an n-element vector, and they can be connected to other squares through weighted links. The whole network is similar in shape to the pointer-based data structure, the only difference being that in place of pointers we have sub-networks which we shall call encoding networks. It turns out that by training the encoding networks to simulate bijections, we end up with something more powerful than pointers. Let us look for example at the sub-network connecting the nodes B, D', E' to B'. If we can train the sub-network to simulate a bijection from $\mathbb{R}^{3n}$ to $\mathbb{R}^n$, B' can be essentially viewed as a pointer to the tree rooted at B as we can simply run the encoding network in reverse order to obtain the values of B, D', E' from B'. In addition, the value of B' itself contains useful information about the tree rooted at B', in fact, it can be understood as a reduced representation of the tree rooted at B. For certain tasks, when B' is referenced, "pointer" chasing may not be required because the value of B' alone may contain sufficient information for proper operation. It is this property that makes it more powerful than pointers, which are really just some meaningless numbers.

In many ways, Hinton's introduction of the concept of a reduced description set the stage for the invention of more realistic architectures.

5.3 Pollack's Recursive Auto-Associative Memory

Appearing in the same volume [Hin90b] is an article by Jordan Pollack [Pol90]. In it, he introduced a connectionist architecture called Recursive Auto-Associative Memory (RAAM) that is capable of automatically developing compact distributed representations of variable-

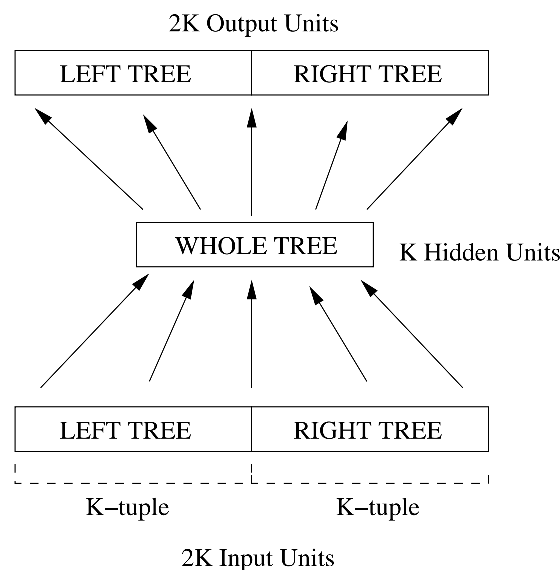


Figure 5.2: Single 2k-k-2k network composed of both the encoding and decoding network.

sized recursive data structures, such as trees and lists, in the form of fixed-width tuples. He reported that "the resulting representations combine apparently immiscible aspects of features, pointers and symbol structures". Pollack's RAAM is one of the first architectures to demonstrate the workability of a reduced-representation-like construct.

Even with the advent of the idea of reduced representations, two fundamentally difficult questions remained in the design of networks that can effectively make use of them. Firstly, it was not entirely clear what a function that can generate reduced descriptions looks like, if indeed one exists. Just exactly how such a function can be simulated using a neural network was a mystery. Secondly, if indeed that can be done, how would the compressed representation look like? It would probably be in the form of real-valued tuples, but it is difficult to envisage how a tuple of real numbers can be crafted to represent all the structural information of a data structure with possibly infinite depth.

With respect to the first question, Pollack simply assumed that one such function can be simulated using a conventional fully-connected feed-forward network with semi-linear units, considering the fact that such networks are universal approximators. For the second question, he developed the strategy of simply letting the network devise its own representations through training. For an example, let us consider the problem of encoding binary trees. Assuming a k-tuple is needed to represent each node of a binary tree, that means a binary tree of depth 1 can be represented using a 2k-tuple. To learn the encoding and decoding mechanisms of one such binary tree, we can use a 2k-k-2k network as shown in Figure 5.2. The two mechanisms can be trained simultaneously using standard back-propagation, with the input values specified as the target values for the output units.

Upon the termination of training, the input and output units will have very similar if

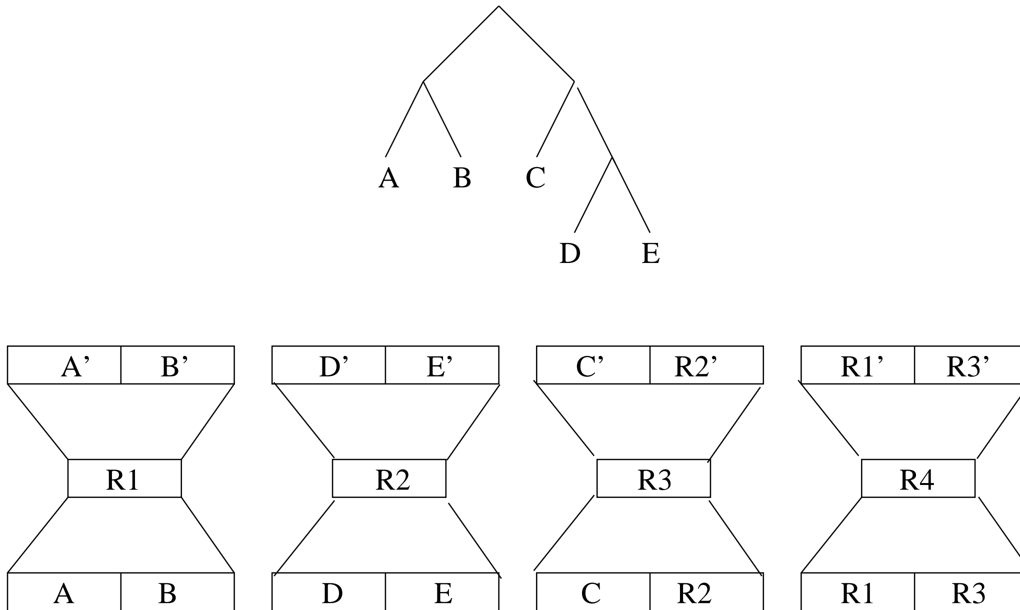


Figure 5.3: To learn both the encoding and decoding mechanisms for the binary tree on top of the figure, the 2k-k-2k network needs to learn the four auto-associations shown. Each rectangle is a K-tuple.

not identical values, and the resulting k-tuple in the hidden unit can be interpreted as a compressed representation of the original depth-1 binary tree. The first part of the network from the input layer to the hidden layer can be seen as a compressor network that simulates the encoding function, while the part of the network from the hidden layer to the output layer can be seen as a reconstructor network that simulates the decoding function.

To find the encoding scheme of general binary trees, we can simply apply this architecture recursively. Figure 5.3 shows the four auto-associations that the network must learn to be able to encode the binary tree ((A,B),(C,(D,E))). The codes for the terminals A, B, C, D and E are supplied to the network, while the codes for the internal nodes $R1$, $R2$, $R3$ and $R4$ must be learned. The learning is a moving-target problem. Weight changes made for one particular association can change the codes for an internal node, this in turn may necessitate further changes because the code for the affected internal node maybe the target for another association. As a result, convergence may take a long time.

On convergence, the differences between the desired and actual outputs will be very close to zero, that is, we have $A' \approx A$, $B' \approx B$, $C' \approx C$, $D' \approx D$, $E' \approx E$, $R1' \approx R1$, $R2' \approx R2$, $R3' \approx R3$. Clearly, $R4$ would in fact be a representation for the whole tree ((A,B),(C,(D,E))). Note that RAAM is a form of distributed representation, hence it suffers from the normal pitfalls associated with such representations. The biggest problem is its inability to represent structures of infinite depth, the network saturates and performs badly when its representational capacity is exceeded.

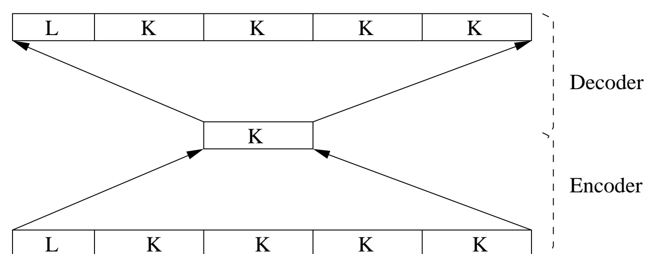


Figure 5.4: The general network architecture of a LRAAM.

Pollack's RAAM is only a hypothetical machine. But it sparked the invention of real machines capable of solving real problems. We look at three architectures motivated by it in the following sections.

5.4 Labelling RAAM

The Labelling RAAM (LRAAM) introduced by Sperduti *et al.* is an extension of the RAAM model that is capable of learning fixed-width distributed representations for labeled variable-sized recursive data structures. Using labeled directed graphs as the most general case of structured data, they show how each node of a labeled directed graph can be represented simply as a $(L + nK)$-tuple, where L is the length of the encoding of a label, n the valence of the graph defined as the maximum outdegree over all the nodes in the graph, and K the length of the encoding of a logical pointer to a node. A graph is simply a collection of $(L + nK)$-tuples whose compressed representation can be learned by applying the LRAAM recursively for each node. Figure 5.4 shows the general architecture of an LRAAM network.

[SSG95] shows how a LRAAM can be used in conjunction with a conventional feed-forward network to learn the classification of logical terms, with good empirical results. The LRAAM and the classifier network were trained simultaneously, thus allowing the classifying task to influence the encoding scheme devised by the LRAAM. This is interesting, as when left alone, the LRAAM is likely to devise schemes that map inputs of similar structures to similar reduced representations, while when interactions exists between the classifier network and the LRAAM, the LRAAM is likely to devise schemes that map inputs of the same class to similar reduced representations. Figure 5.5 shows the network architecture used to perform the classification task.

The form of learning used by Sperduti *et al.* follows the principle of *confluent inference*, where the learning of the distributed representations for structures takes into account the intended task. The other approach is called *transformational inference*, where the unique distributed representations for structures are first computed, then frozen and fed as input to the inference engine.[1]

[1] The terms confluent and transformational inference were introduced by Lonnie Chrisman in [Chr91].

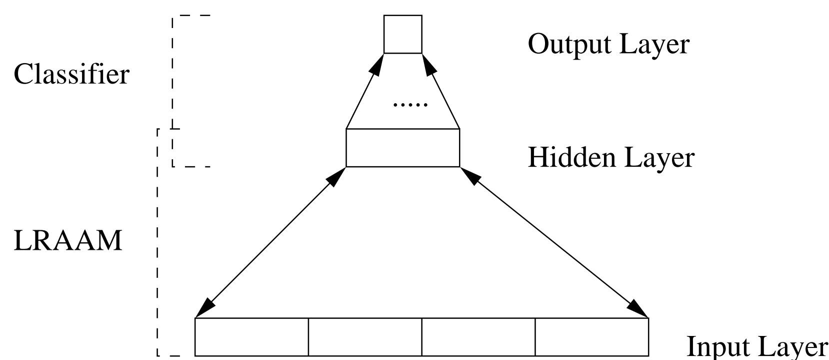


Figure 5.5: The LRAAM-Feedforward network architecture capable of performing classification tasks on terms. Note the double arrow between the input and hidden layers.

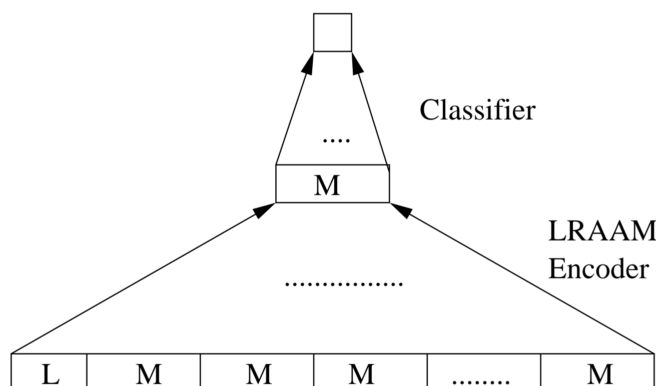


Figure 5.6: The generic form of a folding architecture.

5.5 Folding Architecture

The insights obtained from the LRAAM-based network architectures prompted the later development of the *folding architecture*, described in [GK96]. One important observation from [SSG95] is that inference can be performed even if unique representations of input structures cannot be found, as long as the essential information relevant to the learning task is present. This empirical result suggests that the decoder part of the LRAAM, whose primary role is to ensure unique representations, can be excluded from the network architecture altogether. Figure 5.6 shows the form of a generic folding architecture.

Learning is conducted using an algorithm called *back-propagation through structure* (BPTS). The forward phase of BPTS is the same as in standard back-propagation. The difference lies in the backward phase, where the encoder is unfolded virtually and errors back-propagated from the classifier network. Figure 5.7 (adopted from [GK96]) shows an example of the unfolded encoder network of a term. Note that weights in the different

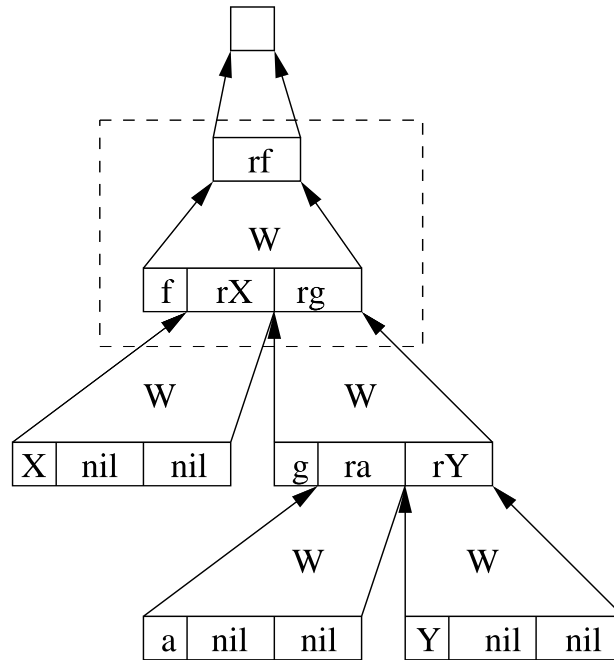


Figure 5.7: The encoder unfolded by the structure $f(X, g(a, Y))$.

copies of the encoder network are the same. Using standard back-propagation, the weights will become different, resulting in a phenomenon called delta-conflicts. Consistency of the weights for these different instances of the same encoder network was maintained by summing the individual gradient contributions of the different instances, and then changing all copies by the same amount at the end of each iteration.

Folding architectures have been used in several real-world applications with good success. Goller applied the technology to classify logical terms in his work on learning search-control heuristics for automated deduction systems [Gol97]. Bianucci *et al.* applied a more advanced version of the folding architecture called *cascade correlation networks for structures* to two problems in chemistry [BMSS00]. They reported that their approach compares favourably against traditional equation-based approaches, and competitively against "ad hoc" multi-layer perceptrons. [MSSB01] conducted some analysis on the internal representations developed by the network on one of the problems.

On the theoretical side, [Ham96] has shown that folding architectures are capable of approximating any mapping between labelled trees and real-valued vector spaces arbitrarily well, *i.e.,* folding architectures are universal approximators from the space of labelled trees to the space of real-valued vectors. [Küc98] shows that there is a direct correspondence between folding architectures and bottom-up tree automata, *i.e.,* for any bottom up tree automaton there is a folding architecture network which simulates it.

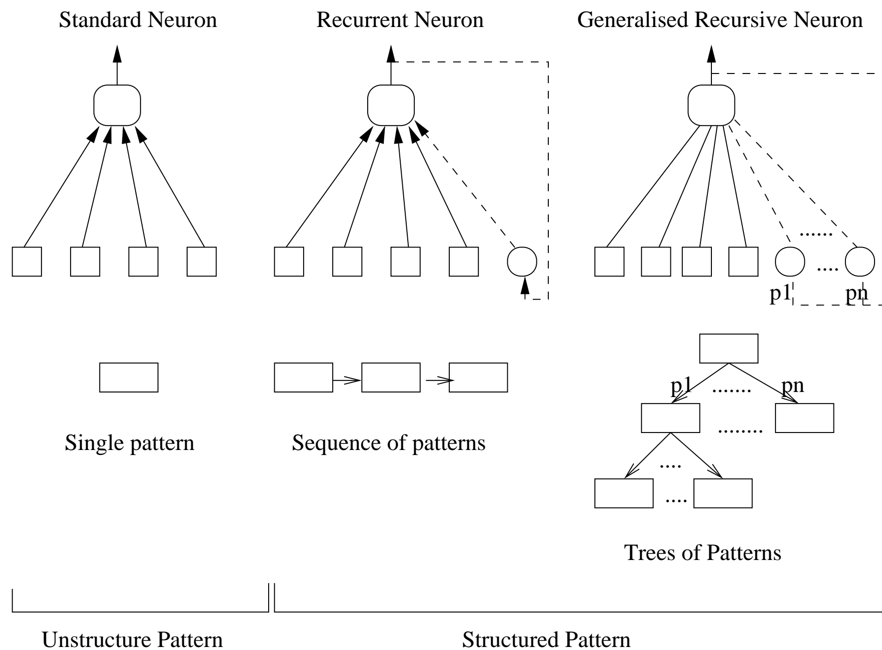


Figure 5.8: Neuron models from different input domains. The standard neuron is suitable for the processing of unstructured patterns, the recurrent neuron for the processing of sequences of patterns, and the generalised recursive neurons for the processing of structured patterns.

5.6 Generalised Recursive Neurons

[SS97] introduced the concept of Generalised Recursive Neurons as a generalisation of standard and recurrent neurons for use in structure encoding networks. They pointed out standard and recurrent neurons are unsuitable as nodes in a structure encoding network because of their limited representation power, and their use often results in complex and unnatural encoding schemes. Generalised recursive neurons was proposed as an alternative. Figure 5.8 (adapted from [SS97]) shows the three forms of neurons and their respective representational capabilities. Note that recurrent neuron is a special case of generalised recursive neurons.

For any labeled graph with a supersource, defined as the vertex from which every other vertex in the graph can be reached, an encoding network can be constructed for it by replicating the same generalised recursive neuron and connecting them according to the topology of the structure. Figure 5.9 illustrates how an encoding network can be constructed for a given graph. The output of the encoding network at the supersource is the neural representation for the graph. The encoding network for cyclic graphs is recurrent. In that case, the graph is represented by the output of the encoding network at a fixed point of the network dynamics.

Sperduti and Starita extended several existing algorithms including back-propagation

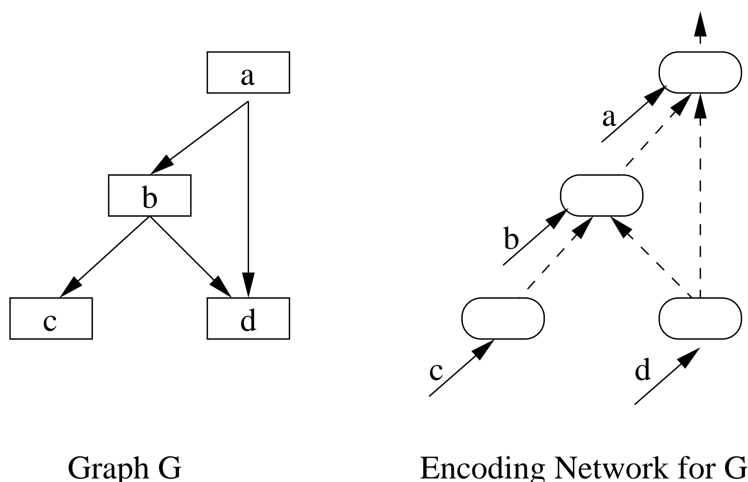


Figure 5.9: A graph and its corresponding encoding network. There is a generalised recursive neuron for every node in the graph. The topology of the network looks similar to the topology of the graph.

through structure and real-time recurrent learning for structure encoding networks composed of generalised recursive neurons. They applied their systems to problems involving classification of logical terms and reported good results.

In many ways, generalised recursive neurons and term reduction networks are complementary technologies. As pointed out in Section 4.2.3, term reduction networks have some difficulties with graphs, in particular cyclic graphs, because there is no obvious way to capture in an explicit fashion all the structural information of a graph using basic terms. This is, however, not a problem for generalised-recursive-neurons-based networks. In fact, generalised recursive neurons are invented exactly for the purpose of solving this class of problems. But generalised recursive neurons cannot handle quite as naturally the many datatypes supported by term reduction networks, particularly datatypes formed using abstractions like sets and multisets. In solving real-world problems, they can be used together as parts of a bigger structural neural network to obtain the best of both worlds.

Chapter 6

Conclusion

A conclusion is the place where you got tired of thinking.
Martin H. Fischer

6.1 Contribution of this thesis

In what way does this thesis contribute to the existing machine learning literature? The answer to this question can best be illustrated with a hypothetical problem:

> You are given 10,000 acyclic graphs with their labels and asked to find a hypothesis to explain the relationship between the graphs and their labels. No background knowledge about the dataset is available. How do you proceed?

This is a tough problem for attribute-value-language-based learners. Since one cannot handle graphs directly, a data preprocessing phase is needed to turn the graphs into feature vectors. But without any background knowledge, this is an almost impossible task. One could try to extract general features about the graphs, *e.g.,* the number of vertices, the number of edges, the connectivity of the graph, the minimum vertex degree, the maximum vertex degree, etc, but chances are that they are all irrelevant to the target function. Without good feature vectors, learning is hard.

The problem is equally tough for symbolic learners because, without any background knowledge, it is hard to tell just exactly what the hypothesis space should look like. Again, one could try to use a hypothesis space with basic and general rules about graphs. But if the target function is of any non-trivial complexity, a real possibility, it is almost inevitable that the learner will just be trapped in a long and futile search.

In term reduction networks, we find a tool that can solve the problem above. And this is the primary contribution of this thesis. We have provided *a method to conduct meaningful learning on structured data in the absence of background knowledge.* This ability to learn without having to make any presuppositions about the target function in advance makes term reduction networks a useful technology to have for many real-world applications like predictive toxicology, where existing domain knowledge is scarce and incomplete.

6.2 Future work

While the empirical results reported in this thesis look positive and promising, much remains to be done on term reduction networks. This final section outlines some potentially interesting directions to take from here onwards.

- One of the biggest limitations of term reduction networks (and neural networks in general) is the incomprehensibility of the concepts they learn. While this has no effect on the trustworthiness of the hypotheses generated, as shown by Baum and Haussler's results [BH89], in several domains of interest like predictive toxicology, it is highly desirable that the concepts learned be translated into a form that can be understood by human experts. There are some work in the literature on rule extraction from neural networks, see for examples [Gal88, Fu91, Hay91]. It would be interesting to investigate the use of some of these techniques to try to extract comprehensible rules out of adapted term reduction networks.

- In the current state, the internal architectures of term reduction networks need to be specified in advance, and the topologies stay fixed through-out training. Topologies chosen this way may not be optimal, and this can result in poor generalisation performance. It would be interesting to introduce adaptive neural network learning techniques (see for examples [BH95, Yao99]) into term reduction networks, so that the best architectures for each component networks can be learned adaptively during training.

- This thesis is primarily an experimental study of term reduction networks. A theoretical study in the future will help answer some very interesting questions that have come up during the course of the present work. The most important of these concerns the approximation capability of term reduction networks. More specifically, are term reduction networks universal approximators? I conjecture this is indeed the case. If we can show that all three reduction networks for basic structures, basic abstractions and basic tuples are universal approximators, then we can prove inductively that term reduction networks are universal approximators. The case for basic structures and basic tuples have been proven by Hammer [Ham96] and Hornik *et al.* [HSW89]. It would be interesting to try to prove the case for basic abstractions in the future.

Appendix A

The (Simple) Mathematics of Back-propagation

This is a standard derivation of the weight update rules for multi-layer feedforward neural networks.

Definition The error function E of the network with respect to the training set is defined as

$$E = \frac{1}{2} \sum_{i}^{N} \sum_{j}^{M} (t_{ji} - o_{ji})^2 \tag{A.1}$$

where N is the total number of individuals, M the number of units in the output layer, and $\vec{t}$ and $\vec{o}$ the target and output vectors of individual i.

Note that equation A.1 is a function of the weights of the network. The derivative of this term with respect to the weights gives the gradient of the error function surface, and we know from elementary calculus that at points where the gradient value is not zero, the negative of this gradient points in the local direction of steepest descent. The central idea of standard back-propagation is to adjust the network weights bit by bit iteratively in the opposite direction of the gradient of steepest ascent, in the hope that on convergence, after a (possibly huge) number of iterations, the network weights will settle onto a minimum on the error surface. That is, on each iteration of the learning process, the individual weights are adjusted according to the following equation:

$$w_{ji} = w_{ji} - \eta \frac{\partial E_d}{\partial w_{ji}} \tag{A.2}$$

where w_{ji} denotes the weight from unit i to unit j in the network and η denotes the learning rate.

The $\frac{\partial E_d}{\partial w_{ji}}$ term in equation A.2 for units in the output unit is different from those in the hidden layers. These will be derived now.

Case 1 We first consider the case of units in the output layer. The only way the weights connecting to one such unit can influence the error value is through the output of the unit.

With that in mind, and using the chain rule, we have

$$\begin{aligned}\frac{\partial E_d}{\partial w_{ji}} &= \frac{\partial E_d}{\partial out_j}\frac{\partial out_j}{\partial w_{ji}} \\ &= \frac{\partial E_d}{\partial out_j}\frac{\partial out_j}{\partial net_j}\frac{\partial net_j}{\partial w_{ji}}\end{aligned} \tag{A.3}$$

where net_j denotes the weighted sum of the inputs to j,

$$net_j = \sum_i w_{ji}x_{ji} \tag{A.4}$$

and out_j denotes the output of unit j,

$$out_j = \sigma(net_j) \tag{A.5}$$

Here, σ denotes a suitable squashing function. In the following analysis, I will assume that σ is the logistic function, defined as follows:

$$\sigma(y) = \frac{1}{1+e^{-y}} \tag{A.6}$$

The logistic function has some mathematical niceties about it. Among them, its derivative with respect to its argument can be expressed in terms of itself as shown below. We will need this result later on.

$$\begin{aligned}\frac{d\sigma(y)}{dy} &= \frac{d\frac{1}{1+e^{-y}}}{dy} \\ &= \frac{\frac{d(1+e^{-y})}{dy}}{(1+e^{-y})^2} \\ &= \frac{e^{-y}}{(1+e^{-y})^2} \\ &= \frac{1}{1+e^{-y}}(1-\frac{1}{1+e^{-y}}) \\ &= \sigma(y)(1-\sigma(y))\end{aligned} \tag{A.7}$$

Equation A.3 can be solved quite easily. To begin, consider the first term in Equation (A.3).

$$\begin{aligned}\frac{\partial E_d}{\partial out_j} &= \frac{\partial \frac{1}{2}\sum_k (t_k - out_k)^2}{\partial out_j} \\ &= \frac{\partial(\frac{1}{2}(t_j - out_j)^2)}{\partial out_j} \\ &= (t_j - out_j)\frac{\partial(t_j - out_j)}{\partial out_j} \\ &= -(t_j - out_j)\end{aligned} \tag{A.8}$$

Next, consider the second term in Equation (A.3). Using Equation (A.7), it can be shown that $\frac{\partial out_j}{\partial net_j}$ is simply $out_j(1 - out_j)$.

$$\begin{aligned}\frac{\partial out_j}{\partial net_j} &= \frac{\partial \sigma(net_j)}{\partial net_j} \\ &= out_j(1 - out_j)\end{aligned} \tag{A.9}$$

Finally, consider the last term in Equation (A.3). $\frac{\partial net_j}{\partial w_{ji}}$ is simply x_{ji}, as shown.

$$\begin{aligned}\frac{\partial net_j}{\partial w_{ji}} &= \frac{\partial \sum_{k \in outputs} w_{jk} x_{jk}}{\partial w_{ji}} \\ &= \frac{\partial (w_{ji} x_{ji})}{\partial w_{ji}} \\ &= x_{ji}\end{aligned} \tag{A.10}$$

Substituting Equations (A.8, A.9, A.10) into Equation (A.3), and combining the result with Equation (A.2), we obtain the weight update rule for output units.

$$\begin{aligned}w_{ji} &= w_{ji} - \eta \frac{\partial E_d}{\partial w_{ji}} \\ &= w_{ji} - \eta \frac{\partial E_d}{\partial out_j} \frac{\partial out_j}{\partial net_j} \frac{\partial net_j}{\partial w_{ji}} \\ &= w_{ji} + \eta (t_j - out_j) out_j (1 - out_j) x_{ji}\end{aligned} \tag{A.11}$$

Case 2 We next consider units in the internal layers. The weights of one such unit affect the error value only indirectly, through the units connected to it. Denoting the set of all

units that take as an input the output value of a unit j by $next(j)$, we have

$$\begin{aligned}
\frac{\partial E_d}{\partial w_{ji}} &= \frac{\partial E_d}{\partial net_j}\frac{\partial net_j}{\partial w_{ji}} \\
&= \sum_{k \in next(j)} \frac{\partial E_d}{\partial net_k}\frac{\partial net_k}{\partial net_j} x_{ji} \\
&= \sum_{k \in next(j)} -\delta_k \frac{\partial net_k}{\partial net_j} x_{ji} \\
&= \sum_{k \in next(j)} -\delta_k \frac{\partial net_k}{\partial out_j}\frac{\partial out_j}{\partial net_j} x_{ji} \\
&= \sum_{k \in next(j)} -\delta_k \frac{\partial w_{kj} out_j}{\partial out_j}\frac{\partial out_j}{\partial net_j} x_{ji} \\
&= \sum_{k \in next(j)} -\delta_k w_{kj} \frac{\partial out_j}{\partial net_j} x_{ji} \\
&= \sum_{k \in next(j)} -\delta_k w_{kj} \frac{\partial \sigma(net_j)}{\partial net_j} x_{ji} \\
&= \sum_{k \in next(j)} -\delta_k w_{kj} out_j (1 - out_j) x_{ji}
\end{aligned} \tag{A.12}$$

Substituting Equation (A.12) into Equation (A.2), we obtain the weight update rule for units in the hidden layers.

$$\begin{aligned}
w_{ji} &= w_{ji} - \eta \frac{\partial E_d}{\partial w_{ji}} \\
&= w_{ji} - \eta \sum_{k \in next(j)} -\delta_k w_{kj} out_j (1 - out_j) x_{ji} \\
&= w_{ji} + \eta out_j (1 - out_j) (\sum_{k \in next(j)} w_{kj} \delta_k) x_{ji}
\end{aligned} \tag{A.13}$$

Appendix B

Sample Individuals in the BST Dataset

B0 Positive
(BNode (BNode (BNode (BNode (Null) 1 (BNode (BNode (BNode (Null) 3 (Null)) 3 (Null)) 3 (Null))) 3 (BNode (Null) 4 (Null))) 4 (BNode (BNode (BNode (BNode (Null) 5 (Null)) 5 (BNode (Null) 6 (Null))) 6 (Null)) 7 (BNode (Null) 8 (Null)))) 8 (BNode (BNode (BNode (BNode (BNode (BNode (Null) 9 (BNode (Null) 10 (Null))) 10 (BNode (BNode (BNode (Null) 11 (Null)) 11 (Null)) 11 (Null))) 12 (BNode (BNode (BNode (BNode (BNode (Null) 13 (Null)) 13 (Null)) 13 (Null)) 13 (Null)) 13 (BNode (Null) 15 (Null)))) 16 (Null)) 16 (Null)) 16 (BNode (BNode (BNode (BNode (Null) 17 (Null)) 17 (Null)) 19 (Null)) 19 (BNode (BNode (Null) 20 (Null)) 20 (Null)))))

B3 Positive
(BNode (BNode (BNode (BNode (BNode (BNode (Null) 3 (Null)) 3 (Null)) 3 (BNode (BNode (BNode (Null) 4 (Null)) 4 (BNode (Null) 5 (Null))) 5 (Null))) 5 (BNode (BNode (BNode (BNode (Null) 6 (Null)) 6 (Null)) 6 (BNode (Null) 7 (BNode (BNode (BNode (BNode (Null) 8 (Null)) 8 (Null)) 9 (Null)) 9 (Null)))) 9 (BNode (Null) 10 (Null)))) 10 (BNode (BNode (BNode (BNode (Null) 12 (Null)) 12 (Null)) 13 (Null)) 13 (BNode (Null) 14 (Null)))) 14 (BNode (BNode (BNode (BNode (BNode (BNode (Null) 15 (Null)) 16 (Null)) 16 (Null)) 18 (BNode (BNode (Null) 19 (Null)) 19 (Null))) 19 (Null)) 19 (BNode (BNode (Null) 20 (Null)) 20 (Null))))

B14 Positive
(BNode (BNode (BNode (Null) 1 (BNode (BNode (BNode (BNode (Null) 3 (Null)) 4 (Null)) 5 (Null)) 6 (BNode (Null) 7 (Null)))) 7 (BNode (BNode (BNode (BNode (Null) 13 (Null)) 13 (BNode (Null) 14 (Null))) 14 (BNode (BNode (BNode (Null) 15 (Null)) 16 (Null)) 16 (BNode (Null) 17 (Null)))) 17 (Null))) 18 (Null))

B20 Positive
(BNode (BNode (Null) 12 (Null)) 15 (BNode (Null) 17 (Null)))

B282 Negative
(BNode (BNode (BNode (Null) 7 (BNode (Null) 9 (Null))) 12 (BNode (BNode (Null) 12 (BNode (Null) 3 (Null))) 3 (BNode (Null) 10 (BNode (Null) 2 (Null))))) 7 (BNode (BNode (BNode (BNode (Null) 14 (Null)) 16 (Null)) 7 (Null)) 20 (BNode (BNode (BNode (Null) 10 (Null)) 8 (Null)) 20 (BNode (Null) 3 (Null)))))

B496 Negative
(BNode (BNode (BNode (BNode (Null) 9 (Null)) 2 (BNode (BNode (BNode (Null) 11 (BNode (Null) 7 (Null))) 20 (BNode (Null) 3 (Null))) 20 (BNode (Null) 13 (Null)))) 6 (BNode (BNode (Null) 12 (Null)) 11 (Null))) 15 (BNode (BNode (BNode (Null) 9 (BNode (Null) 1 (Null))) 14 (BNode (Null) 16 (Null))) 5 (BNode (Null) 12 (Null))))

B497 Negative
(BNode (BNode (BNode (BNode (BNode (Null) 5 (Null)) 5 (BNode (Null) 1 (BNode (Null) 6 (Null)))) 6 (BNode (BNode (Null) 19 (BNode (Null) 7 (Null))) 8 (BNode (Null) 19 (Null)))) 19 (BNode (BNode (BNode (Null) 16 (Null)) 17 (Null)) 20 (BNode (Null) 9 (Null)))) 7 (BNode (BNode (BNode (BNode (Null) 12 (Null)) 13 (BNode (Null) 20 (Null))) 13 (BNode (BNode (BNode (Null) 1 (Null)) 19 (Null)) 2 (BNode (Null) 18 (Null)))) 20 (BNode (BNode (BNode (Null) 8 (Null)) 18 (BNode (BNode (BNode (Null) 11 (Null)) 12 (Null)) 12 (Null))) 20 (BNode (BNode (Null) 5 (Null)) 3 (Null)))))

B498 Negative
(BNode (BNode (BNode (BNode (BNode (Null) 19 (BNode (Null) 7 (Null))) 2 (Null)) 6 (BNode (BNode (Null) 8 (Null)) 17 (BNode (BNode (Null) 14 (Null)) 13 (BNode (Null) 20 (Null))))) 1 (BNode (BNode (BNode (Null) 6 (Null)) 5 (BNode (Null) 14 (BNode (Null) 13 (Null)))) 3 (BNode (BNode (Null) 1 (Null)) 12 (BNode (Null) 6 (Null))))) 1 (BNode (BNode (BNode (BNode (BNode (Null) 6 (Null)) 16 (Null)) 2 (BNode (Null) 1 (BNode (Null) 6 (BNode (Null) 5 (Null))))) 6 (BNode (BNode (BNode (Null) 17 (Null)) 18 (Null)) 19 (BNode (Null) 17 (Null)))) 1 (BNode (BNode (BNode (Null) 13 (BNode (Null) 3 (Null))) 18 (BNode (Null) 12 (BNode (Null) 11 (Null)))) 8 (BNode (Null) 13 (BNode (BNode (Null) 10 (Null)) 2 (Null))))))

B499 Negative
(BNode (BNode (BNode (BNode (BNode (BNode (BNode (Null) 11 (Null)) 1 (Null)) 4 (BNode (BNode (Null) 3 (Null)) 5 (Null))) 17 (BNode (Null) 1 (Null))) 11 (BNode (BNode (BNode (BNode (Null) 20 (Null)) 13 (Null)) 15 (Null)) 3 (BNode (BNode (Null) 17 (Null)) 14 (Null)))) 7 (BNode (BNode (BNode (BNode (BNode (Null) 19 (Null)) 12 (Null)) 8 (BNode (BNode (Null) 13 (Null)) 7 (BNode (Null) 15 (Null)))) 16 (BNode (BNode (Null) 11 (Null)) 15 (BNode (Null) 8 (Null))))) 12 (BNode (BNode (BNode (Null) 2 (BNode (Null) 3 (Null))) 5 (BNode (BNode (Null) 12 (Null)) 19 (Null))) 20 (BNode (BNode (BNode (Null) 12 (Null)) 5 (Null)) 5 (Null)))))

Bibliography

[AB99] Martin Anthony and Peter L. Bartlett. *Neural Network Learning: Theoretical Foundations*. Cambridge University Press, 1999.

[And86] Peter B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*. Academic Press, 1986.

[BGCL00] Antony F. Bowers, Christophe Giraud-Carrier, and John W. Lloyd. Classification of individuals with complex structure. In P. Langley, editor, *Machine Learning: Proceedings of the Seventeenth International Conference (ICML2000)*, pages 81–88. Morgan Kaufmann, 2000.

[BGCL01] Antony F. Bowers, Christophe Giraud-Carrier, and John W. Lloyd. A knowledge representation framework for inductive learning. Available at `http://discus.anu.edu.au/~jwl/kr.ps.gz` 2001.

[BH89] Eric B. Baum and David Haussler. What size net gives valid generalization. *Neural Computation*, 1:151–160, 1989.

[BH95] Karthik Balakrishnan and Vasant Honavar. Evolutionary Design of Neural Architectures: A Preliminary Taxonomy and Guide to Literature. Technical report, Department of Computer Science, Iowa State University, Ames, Iowa, 1995.

[Bis95] Christopher M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, 1995.

[BMSS00] Anna Maria Bianucci, Alessio Micheli, Alessandro Sperduti, and Antonina Starita. Application of cascade correlation networks for structures to chemistry. *Journal of Applied Intelligence*, 12 (1/2):117–147, 2000.

[Bon70] Mikhail M. Bongard. *Pattern Recognition*. Spartan Books, 1970.

[BR98] Hendrik Blockeel and Luc De Raedt. Top-down induction of first-order logical decision trees. *Artificial Intelligence*, 101(1–2):285–297, 1998.

[Bra90] Ivan Bratko. *Prolog: Programming for Artificial Intelligence*. Addison-Wesley, 2nd edition, 1990.

[Chr91] Lonnie Chrisman. Learning recursive distributed representations for holistic computation. *Connection Science*, 3(4):345–366, 1991.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[Cop71] Irving Copi. *The Theory of Logical Types, Monographs in modern logic series.* Routledge and Kegan Paul, 1971.

[CST00] Nello Cristianini and John Shawe-Taylor. *An Introduction to Support Vector Machines and Other Kernel-based Learning Methods.* Cambridge University Press, Cambridge, 2000.

[DLLP97] Thomas G. Dietterich, Richard H. Lathrop, and Tomás Lozano-Pérez. Solving the multiple instance problem with axis-parallel rectangles. *Artificial Intelligence*, 89:31–71, 1997.

[Eas] Home page of East-West Challenge.
`http://www.gmd.de/ml-archive/ILP/public/data/east_west/`.

[Fah89] Scott E. Fahlman. Faster-learning variations on back-propagation: an empirical study. In D. Touretzky, G. Hinton, and T. Sejnowski, editors, *Proceedings of the 1988 Connectionist Models Summer School, Pittsburg*, pages 38–51, San Mateo, CA, 1989. Morgan Kaufmann.

[FGCL98] Peter A. Flach, Christophe Giraud-Carrier, and John W. Lloyd. Strongly typed inductive concept learning. In D. Page, editor, *Proceedings of the 8th International Conference on Inductive Logic Programming*, volume 1446, pages 185–194. Springer-Verlag, 1998.

[FL90] Scott E. Fahlman and Christian Lebiere. The cascade-correlation learning architecture. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems*, volume 2, pages 524–532, Denver 1989, 1990. Morgan Kaufmann, San Mateo.

[Fla00] Peter A. Flach. The use of functional and logic languages in machine learning. In Maria Alpuente, editor, *Ninth International Workshop on Functional and Logic Programming (WFLP2000)*, pages 225–237. Universidad Politecnica de Valencia, 2000.

[Fu91] LiMin Fu. Rule learning by searching on adapted nets. In *AAAI, Vol. 2*, pages 590–595, 1991.

[Gal88] Stephen I. Gallant. Connectionist expert systems. *Communications of the ACM*, 31(2):152–168, 1988.

[GK96] Christoph Goller and Andreas Küchler. Learning task-independent distributed representations by backpropagation through structure. In *Proceedings of the International Conference on Neural Networks (ICNN-96)*, pages 347–352, 1996.

[Gol97] Christoph Goller. *A Connectionist Approach for Learning Search-Control Heuristics for Automated Deduction Systems.* PhD thesis, Fakultät für Informatik der Technischen Universität München, 1997.

[Ham96] Barbara Hammer. Universal approximation of mappings on structured objects using the folding architecture. Technical Report 183, Universitat Osnabruck, Fachbereich Informatik, 1996.

[Has] The Haskell home page. `http:://www.haskell.org`.

[Hay91] Yoichi Hayashi. A neural expert system with automated extraction of fuzzy if-then rules. In Richard P. Lippmann, John E. Moody, and David S. Touretzky, editors, *Advances in Neural Information Processing Systems*, volume 3, pages 578–584. Morgan Kaufmann Publishers, Inc., 1991.

[Hay99] Simon Haykin. *Neural Networks: A Comprehensive Foundation.* Prentice Hall, 1999.

[Hea96] Michael T. Heath. *Scientific Computing - An Introductory Survey.* McGraw-Hill, 1996.

[Hen50] Leon Henkin. Completeness in the theory of types. *Journal of Symbolic Logic*, 15(2):81–91, 1950.

[Hin90a] Geoffrey E. Hinton. Mapping part-whole hierarchies into connectionist networks. *Artificial Intelligence*, 46:77–106, 1990.

[Hin90b] Geoffrey E. Hinton. Special Issue: Connectionist symbol processing. *Artificial Intelligence*, 46(1/2), 1990.

[HK01] Jiawei Han and Micheline Kamber. *Data Mining: Concepts and Techniuqes.* Morgan Kaufmann Publishers, 2001.

[HKKe01] Christoph Helma, Ross King, Stefan Kramer, and Ashwin Srinivasan (editors). *Proceedings of PKDD01 Workshop on The Predictive Toxicology Challenge.* 2001. Available at `http://falcon.informatik.uni-freiburg.de/~ml/ptc/`.

[HSW89] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.

[Jac88] Robert A. Jacobs. Increased rates of convergence through learning rate adaptation. *Neural Networks*, 1(4):295–307, 1988.

[KG96] Andreas Küchler and Christoph Goller. Inductive learning in symbolic domains using structure-driven recurrent neural networks. In *KI - Kunstliche Intelligenz*, pages 183–197, 1996.

[Knu97] Donald E. Knuth. *The Art of Computer Programming, Volume 1, Fundamental Algorithms.* Addison-Wesley, 3rd edition, 1997.

[Koh95] Ron Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 1137–1145, 1995.

[KP91] John F. Kolen and Jordan B. Pollack. Back propagation is sensitive to initial conditions. In Richard P. Lippmann, John E. Moody, and David S. Touretzky, editors, *Advances in Neural Information Processing Systems*, volume 3, pages 860–867. Morgan Kaufmann Publishers, Inc., 1991.

[Küc98] Andreas Küchler. On the correspondence between neural folding architectures and tree automata. Technical Report 98-06, Faculty of Computer Science, University of Ulm, 1998.

[Llo95] John W. Lloyd. Declarative programming in Escher. Technical Report CSTR-95-013, Department of Computer Science, Bristol University, 1995.

[Llo99] John W. Lloyd. Programming in an integrated functional and logic language. *Journal of Functional and Logic Programming*, 3, 1999.

[Llo00] John W. Lloyd. Neural networks for structured data. Unpublished manuscript, 2000.

[Llo01] John W. Lloyd. Knowledge representation, computation, and learning in higher-order logic. In preparation, 2001.

[MB88] Stephen H. Muggleton and W. Buntine. Machine invention of first-order predicates by inverting resolution. In *Proc. Fifth International Conference on Machine Learning*, pages 339–352, San Mateo, CA, 1988. Morgan Kaufmann.

[Mit96] John C. Mitchell. *Foundations for Programming Languages.* MIT Press, Cambridge, MA, 1996.

[Mit97] Tom M. Mitchell. *Machine Learning.* McGraw-Hill, 1997.

[ML77] Ryszard S. Michalski and J. Larson. Inductive inference of VL decision rules. In *Proceedings of the Workshop in Pattern-Directed Inference Systems*, pages 34–44, 1977.

[Moo94] John Moody. Prediction risk and architecture selection for neural networks. In V. Cherkassky, J. H. Friedman, and H. Wechsler, editors, *From Statistics to Neural Networks: Theory and Pattern Recognition Applications*. Springer, NATO ASI Series F, 1994.

[MP69] Marvin L. Minsky and S. Papert. *Perceptrons*. MIT Press, Cambridge, 1969.

[MR94] Stephen Muggleton and Luc De Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19,20:629–679, 1994.

[MSSB01] Alessio Micheli, Alessandro Sperduti, Antonina Starita, and Anna Maria Bianucci. Analysis of the internal representations developed by neural networks for structures applied to quantitative structure-activity relationship studies of benzodiazepines. *Journal of Chemical Information and Computer Sciences*, 41 (1):202–218, 2001.

[Mug95] Stephen Muggleton. Inverse entailment and Progol. *New Generation Computing*, 13:245–286, 1995.

[Mug99] Stephen Muggleton. Inductive Logic Programming. In *The MIT Encyclopedia of the Cognitive Sciences (MITECS)*. MIT Press, 1999.

[NCdW97] S. Nienhuys-Cheng and R. de Wolf. *Foundations of Inductive Logic Programming*, volume LNAI 1228. Springer-Verlag, 1997.

[NLS01] Kee Siong Ng, John W. Lloyd, and Andrew W. Slater. Predictive toxicology using a decision-tree learner. In *The 2000-1 Predictive Toxicology Challenge Workshop, 5th European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD 2001)*, 2001.

[Pen89] Roger Penrose. *The Emperor's New Mind*. Oxford University Press, 1989.

[PF01] Foster Provost and Tom Fawcett. Robust classification for imprecise environments. *Machine Learning Journal*, 42(3), 2001.

[PK92] Michael J. Pazzani and Dennis Kibler. The utility of knowledge in inductive learning. *Machine Learning*, 5:239–266, 1992.

[Pol90] Jordan B. Pollack. Recursive distributed representations. *Artificial Intelligence*, 46(1-2):77–106, 1990.

[PTC01] Home page of PTC.
`http://falcon.informatik.uni-freiburg.de/~ml/ptc`, 2001.

[PTVF92] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, 2nd edition, 1992.

[Qui86] J. Ross Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.

[Qui90] J. Ross Quinlan. Learning logical definitions from relations. *Machine Learning*, 5(3):239–266, 1990.

[Qui96] J. Ross Quinlan. Learning first-order definitions of functions. *Journal of Artificial Intelligence Research*, 5:139–161, 1996.

[RB93] Martin Riedmiller and Heinrich Braun. A direct adaptive method for faster backpropagation learning: The Rprop algorithm. In *In Proccedings of the 1993 IEEE International Conference on Neural Networks*, volume 1, pages 586–591, 1993.

[Ric01] Rod Rickards. Personal correspondence, 2001.

[RM99] Russell D. Reed and Robert J. Marks. *Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks*. MIT Press, 1999.

[RR00] Jan Ramon and Luc De Raedt. Multi instance neural networks. In *Proceedings of the ICML-2000 Workshop on Attribute-Value and Relational Learning*, 2000.

[SA90] Fernando M. Silva and Luis B. Almeida. Speeding up backpropagation. In R. Eckmiller, editor, *Advanced Neural Computers*, pages 151–158. Elsevier North Holland, Amsterdam, 1990.

[Sar95] Warren Sarle. Stopped training and other remedies for overfitting. In *Proceedings of the 27th Symposium on the Interface*, pages 1–10, 1995.

[Sar00] Warren Sarle. Neural network faq, 2000. Sarle, W.S., ed. (2000), Neural Network FAQ, periodic posting to the Usenet newsgroup comp.ai.neural-nets, URL: ftp://ftp.sas.com/pub/neural/FAQ.html. Referenced 27.3.2000.

[Sha92] Jude W. Shavlik. A framework for combining symbolic and neural learning. Technical Report 1123, Computer Sciences Department, University of Wisconsin-Madison, 1992.

[SS97] Alessandro Sperduti and Antonina Starita. Supervised neural networks for the classification of structures. *IEEE Transactions on Neural Networks*, 8(3):714–735, 1997.

[SSG95] Alessandro Sperduti, Antonina Starita, and Christoph Goller. Learning distributed representations for the classification of terms. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 509–515, 1995.

[TF97] Georg Thimm and Emile Fiesler. High-order and multilayer perceptron initialization. *IEEE Transactions on Neural Networks*, 8(2):249–259, 1997.

[Tol90] Tom Tollenaere. SuperSAB: Fast adaptive back propagation with good scaling properties. *Neural Networks*, 3(5):561–573, 1990.

[TS93] Geoffrey G. Towell and Jude W. Shavlik. Knowledge-based artificial neural networks,. *Artificial Intelligence*, 70:119–165, 1993.

[Val84] Leslie G. Valiant. A theory of the learnable. *Communications of the Association for Computing Machinery*, 27(11):1134–1142, 1984.

[VMR+88] Thomas P. Vogl, J. K. Mangis, A. K. Rigler, W. T. Zink, and D. L. Alkon. Accelerating the convergence of the back-propagation method. *Biological Cybernetics*, 59:257–263, 1988.

[Wol93] David Wolfram. *The Clausal Theory of Types.* Cambridge University Press, 1993.

[Yao99] Xin Yao. Evolving artificial neural networks. *Proceedings of the IEEE*, 87 (9):1423–1447, 1999.